2

Exact Matching: Classical Comparison-Based Methods

2.1. Introduction

This chapter develops a number of classical comparison-based matching algorithms for the exact matching problem. With suitable extensions, all of these algorithms can be implemented to run in linear worst-case time, and all achieve this performance by preprocessing pattern P. (Methods that preprocess T will be considered in Part II of the book.) The original preprocessing methods for these various algorithms are related in spirit but are quite different in conceptual difficulty. Some of the original preprocessing methods are quite difficult.[1] This chapter does not follow the original preprocessing methods but instead exploits fundamental preprocessing, developed in the previous chapter, to implement the needed preprocessing for each specific matching algorithm.

Also, in contrast to previous expositions, we emphasize the Boyer–Moore method over the Knuth-Morris-Pratt method, since Boyer–Moore is the practical method of choice for exact matching. Knuth-Morris-Pratt is nonetheless completely developed, partly for historical reasons, but mostly because it generalizes to problems such as real-time string matching and matching against a set of patterns more easily than Boyer–Moore does. These two topics will be described in this chapter and the next.

2.2. The Boyer–Moore Algorithm

As in the naive algorithm, the Boyer–Moore algorithm successively aligns P with T and then checks whether P matches the opposing characters of T. Further, after the check is complete, P is shifted right relative to T just as in the naive algorithm. However, the Boyer–Moore algorithm contains three clever ideas not contained in the naive algorithm: the right-to-left scan, the bad character shift rule, and the good suffix shift rule. Together, these ideas lead to a method that typically examines fewer than $m + n$ characters (an expected sublinear-time method) and that (with a certain extension) runs in linear worst-case time. Our discussion of the Boyer–Moore algorithm, and extensions of it, concentrates on ***provable*** aspects of its behavior. Extensive experimental and practical studies of Boyer–Moore and variants have been reported in [229], [237], [409], [410], and [425].

2.2.1. Right-to-left scan

For any alignment of P with T the Boyer–Moore algorithm checks for an occurrence of P by scanning characters from *right to left* rather than from left to right as in the naive

[1] Sedgewick [401] writes "Both the Knuth-Morris-Pratt and the Boyer–Moore algorithms require some complicated preprocessing on the pattern that is difficult to understand and has limited the extent to which they are used". In agreement with Sedgrwick, I still do not understand the original Boyer–Moore preprocessing method for the *strong* good suffix rule,

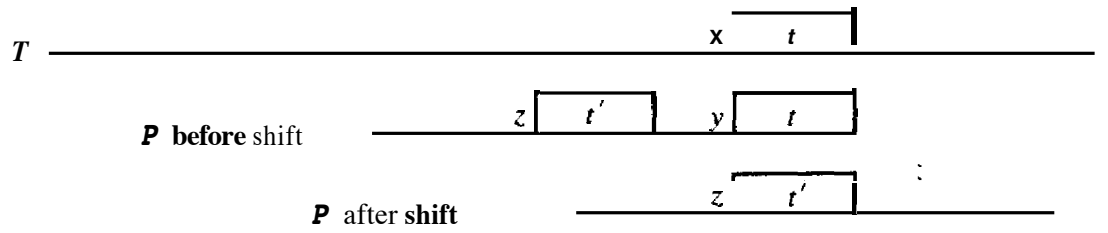

Figure 2.1: Good suffix shift rule, where character x of *T* mismatches with character *y* of P. Characters *y* and *z* of P are guaranteed to be distinct by the good suffix rule, so *z* has a chance of matching x.

good *suffix rule*. The original preprocessing method [278] for the strong good suffix rule is generally considered quite difficult and somewhat mysterious (although a weaker version of it is easy to understand). In fact, the preprocessing for the strong rule was given incorrectly in [278] and corrected, without much explanation, in [384]. Code based on [384] is given without real explanation in the text by Baase [32], but there are no published sources that try to fully explain the method.[2] Pascal code for strong preprocessing, based on an outline by Richard Cole [107], is shown in Exercise 24 at the end of this chapter.

In contrast, the fundamental preprocessing of P discussed in Chapter 1 makes the needed preprocessing very simple. That is the approach we take here. The *strong good suffix rule* is:

> Suppose for a given alignment of P and T, a substring t of T matches a suffix of P, but a mismatch occurs at the next comparison to the left. Then find, if it exists, the right-most copy t' of t in P such that t' is not a suffix of P and *the character to the left of* t' *in* P *differs from the character to the left of* t *in* P. Shift P to the right so that substring t' in P is below substring t in T (see Figure 2.1). If t' does not exist, then shift the left end of P past the left end of t in T by the least amount so that a prefix of the shifted pattern matches a suffix of t in T. If no such shift is possible, then shift P by n places to the right. If an occurrence of P is found, then shift P by the least amount so that a *proper* prefix of the shifted P matches a suffix of the occurrence of P in T. If no such shift is possible, then shift P by n places, that is, shift P past t in T.

For a specific example consider the alignment of P and T given below:

```
          0         1
          123456789012345678
T:        prstabstubabvqxrst
                    *
P:          qcabdabdab
            1233567890
```

When the mismatch occurs at position 8 of P and position 10 of T, $t = ab$ and t' occurs in P starting at position 3. Hence P is shifted right by six places, resulting in the following alignment:

[2] A recent plea appeared on the internet newsgroup comp.theory:

> I am looking for an elegant (easily understandable) proof of correctness for a part of the Buyer-Moore string matching algorithm. The difficutt-to-prove part here is the algorithm that computes the dd_2 (good-suffix)table. I didn't find much of an understandable proof yet, so I'd much appreciate any help!

Extended bad character rule

The bad character rule is a useful heuristic for mismatches near the right end of P, but it has no effect if the mismatching character from T occurs in P to the right of the mismatch point. This may be common when the alphabet is small and the text contains many similar, but not exact, substrings. That situation is typical of DNA, which has an alphabet of size four, and even protein, which has an alphabet of size twenty, often contains different regions of high similarity. In such cases, the following ***extended bad character*** rule is more robust:

> When a mismatch occurs at position i of P and the mismatched character in T is **x**, then shift P to the right so that the closest x to the left of position i in P is below the mismatched ***x*** in T.

Because the extended rule gives larger shifts, the only reason to prefer the simpler rule is to avoid the added implementation expense of the extended rule. The simpler rule uses only $O(|\Sigma|)$ space (Σ is the alphabet) for array R, and one table lookup for each mismatch. As we will see, the extended rule can be implemented to take only $O(n)$ space and at most one extra step per character comparison. That amount of added space is not often a critical issue, but it is an empirical question whether the longer shifts make up for the added time used by the extended rule. The original Boyer–Moore algorithm only uses the simpler bad character rule.

Implementing the extended bad character rule

We preprocess P so that the extended bad character rule can be implemented efficiently in both time and space. The preprocessing should discover, for each position i in P and for each character x in the alphabet, the position of the closest occurrence of x in P to the left of i. The obvious approach is to use a two-dimensional array of size n by $|\Sigma|$ to store this information. Then, when a mismatch occurs at position i of P and the mismatching character in T is x, we look up the (i, x) entry in the array. The lookup is fast, but the size of the array, and the time to build it, may be excessive. A better compromise, below, is possible.

During preprocessing, scan P from right to left collecting, for each character x in the alphabet, a list of the positions where x occurs in P. Since the scan is right to left, each list will be in decreasing order. For example, if P = abacbabc then the list for character a is 6, 3, 1. These lists are accumulated in $O(n)$ time and of course take only $O(n)$ space. During the search stage of the Boyer–Moore algorithm if there is a mismatch at position i of P and the mismatching character in T is **x**, scan **x's** list from the top until we reach the first number less than i or discover there is none. If there is none then there is no occurrence of **x** before i, and all of P is shifted past the x in T. Otherwise, the found entry gives the desired position of ***x***.

After a mismatch at position i of P the time to scan the list is at most n − i, which is roughly the number of characters that matched. So in worst case, this approach at most doubles the running time of the Boyer–Moore algorithm. However, in most problem settings the added time will be vastly less than double. One could also do binary search on the list in circumstances that warrant it.

2.2.3. The (strong) good suffix rule

The bad character rule by itself is reputed to be highly effective in practice, particularly for English text [229], but proves less effective for small alphabets and it does not lead to a linear worst-case running time. For that, we introduce another rule called the ***strong***

For example, if P = cabdabdab, then $N_3(P) = 2$ and $N_6(P) = 5$.

Recall that $Z_i(S)$ is the length of the longest substring of S that starts at i and matches a prefix of S. Clearly, N is the reverse of Z, that is, if P^r denotes the string obtained by reversing P, then $N_i(P) = Z_{n-j+1}(P^r)$. Hence the $N_i(P)$ values can be obtained in $O(n)$ time by using Algorithm Z on P'. The following theorem is then immediate.

Theorem 2.2.2. *$L(i)$ is the largest index j less than* n such thar $N_j(P) \geq |P[i..n]|$ *(which is $n-i+1$). $L'(i)$ is the largest index j less than n such that $N_j(P) = |P[i..n]| = (n-i+1)$.*

Given Theorem 2.2.2, it follows immediately that all the $L'(i)$ values can be accumulated in linear time from the N values using the following algorithm:

Z-based Boyer–Moore

```
for i := 1 to n do L'(i) := 0;
for j := 1 to n - 1 do
    begin
    i := n - N_j(P) + 1;
    L'(i) := j;
    end;
```

The $L(i)$ values (if desired) can be obtained by adding the following lines to the above pseudocode:

```
L(2) := L'(2);
for i := 3 to n do L(i) := max[L(i - 1), L'(i)];
```

Theorem 2.2.3. *The* above *method correctly* computes *the L values.*

PROOF $L(i)$ marks the right end-position of the right-most substring of P that matches $P[i..n]$ and is not a suffix of $P[1..n]$. Therefore, that substring begins at position $L(i)-n+i$, which we will denote by j. We will prove that $L(i) = \max[L(i-1), L'(i)]$ by considering what character j − 1 is. First, if $j = 1$ then character $j - 1$ doesn't exist, so L(i − 1) = 0 and $L'(i) = 1$. So suppose that j > 1. If character j − 1 equals character i − 1 then $L(i) = L(i-1)$. If character $j - 1$ does not equal character i − 1 then $L(i) = L'(i)$. Thus, in all cases, $L(i)$ must either be $L'(i)$ or $L(i-1)$.

However, $L(i)$ must certainly be greater than or equal to both $L'(i)$ and L(i − 1). In summary, $L(i)$ must either be $L'(i)$ or L(i − 1), and yet it must be greater or equal to both of them; hence $L(i)$ must be the maximum of $L'(i)$ and $L(i-1)$. □

Final preprocessing detail

The preprocessing stage must also prepare for the case when $L'(i) = 0$ or when an occurrence of P is found. The following definition and theorem accomplish that.

Definition Let $l'(i)$ denote the length of the largest suffix of $P[i..n]$ that is also a prefix of P, if one exists. If none exists, then let $l'(i)$ be zero.

Theorem 2.2.4. *$l'(i)$ equals the* largest $j \leq |P[i..n]|$, which is n − i + 1, *such that $N_j(P) = j$.*

We leave the proof, as well as the problem of how to accumulate the $l'(i)$ values in linear time, as a simple exercise. (Exercise 9 of this chapter)

```
          0         1
          123456789012345678
T:        prstabstubabvqxrst
P:              qcabdabdab
```

Note that the extended bad character rule would have shifted P by only one place in this example.

Theorem 2.2.1. *The* use of *the good suffix rule never shifts P past an occurrence in T.*

PROOF Suppose the right end of P is aligned with character k of T before the shift, and suppose that the good suffix rule shifts P so its right end aligns with character $k' > k$. Any occurrence of P ending at a position l strictly between k and k' would immediately violate the selection rule for k', since it would imply either that a closer copy of t occurs in P or that a longer prefix of P matches a suffix of t. □

The original published Boyer–Moore algorithm [75] uses a simpler, weaker, version of the good suffix rule. That version just requires that the shifted P agree with the t and does not specify that the next characters to the left of those occurrences of t be different. An explicit statement of the weaker rule can be obtained by deleting the italics phrase in the first paragraph of the statement of the strong good suffix rule. In the previous example, the weaker shift rule shifts P by three places rather than six. When we need to distinguish the two rules, we will call the simpler rule the weak good suffix rule and the rule stated above the strong good suffix rule. For the purpose of proving that the search part of Boyer–Moore runs in linear worst-case time, the weak rule is not sufficient, and in this book the strong version is assumed unless stated otherwise.

2.2.4. Preprocessing for the good suffix rule

We now formalize the preprocessing needed for the Boyer–Moore algorithm.

Definition For each i, $L(i)$ is the largest position less than n such that string $P[i..n]$ matches a suffix of $P[1..L(i)]$. $L(i)$ is defined to be zero if there is no position satisfying the conditions. For each i, $L'(i)$ is the largest position less than n such that string $P[i..n]$ matches a suffix of $P[1..L'(i)]$ and such that the character preceding that suffix is not equal to $P(i-1)$. $L'(i)$ is defined to be zero if there is no position satisfying the conditions.

For example, if P = *cabdabdab*, then $L(8) = 6$ and $L'(8) = 3$.

$L(i)$ gives the right end-position of the right-most copy of $P[i..n]$ that is not a suffix of P, whereas $L'(i)$ gives the right end-position of the right-most copy of $P[i..n]$ that is not a suffix of P, with the stronger, added condition that its preceding character is unequal to $P(i-1)$. So, in the strong-shift version of the Boyer–Moore algorithm, if character $i-1$ of P is involved in *a* mismatch and $L'(i) > 0$, then P is shifted right by $n - L(i)$ positions. *The* result is that if the right end of P was aligned with position k of T before the shift, then position $L(i)$ is now aligned with position k.

During the preprocessing stage of the Boyer–Moore algorithm $L'(i)$ *(and* $L(i)$, if desired) will be computed for each position i in P. This is done in $O(n)$ time via the following definition and theorem.

Definition For string P, $N_j(P)$ is the length of the longest *suffix* of the substring $P[1..j]$ that is also a *suffix* of the full string P.

Boyer–Moore method has a worst-case running time of $O(m)$ provided that the pattern does not appear in the text. This was first proved by Knuth, Moms, and Pratt [278], and an alternate proof was given by Guibas and Odlyzko [196]. Both of these proofs were quite difficult and established worst-case time bounds no better than $5m$ comparisons. Later, Richard Cole gave a much simpler proof [108] establishing a bound of 4m comparisons and also gave a difficult proof establishing a tight bound of 3m comparisons. We will present Cole's proof of 4m comparisons in Section 3.2.

When the pattern does appear in the text then the original Boyer–Moore method runs in $\Theta(nm)$ worst-case time. However, several simple modifications to the method correct this problem, yielding an $O(m)$ time bound in all cases. The first of these modifications was due to Galil [168]. After discussing Cole's proof, in Section 3.2, for the case that P doesn't occur in T, we use a variant of Galil's idea to achieve the linear time bound in all cases.

At the other extreme, if we only use the bad character shift rule, then the worst-case running time is $O(nm)$, but assuming randomly generated strings, the expected running time is sublinear. Moreover, in typical string matching applications involving natural language text, a sublinear running time is almost always observed in practice. We won't discuss random string analysis in this book but refer the reader to [184].

Although Cole's proof for the linear worst case is vastly simpler than earlier proofs, and is important in order to complete the full story of Boyer–Moore, it is not trivial. However, a fairly simple extension of the Boyer–Moore algorithm, due to Apostolico and Giancarlo [26], gives a "Boyer–Moore–like" algorithm that allows a fairly direct proof of a 2m worst-case bound on the number of comparisons. The Apostolico–Giancarlo variant of Boyer–Moore is discussed in Section 3.1.

2.3. The Knuth-Morris-Pratt algorithm

The best known linear-time algorithm for the exact matching problem is due to Knuth, Morris, and Pratt [278]. Although it is rarely the method of choice, and is often much inferior in practice to the Boyer–Moore method (and others), it can be simply explained, and its linear time bound is (fairly) easily proved. The algorithm also forms the basis of the well-known Aho–Corasick algorithm, which efficiently finds all occurrences in a text of any pattern from a set of patterns.[3]

2.3.1. The Knuth-Morris-Pratt shift idea

For a given alignment of P with T, suppose the naive algorithm matches the first i characters of P against their counterparts in T and then mismatches on the next comparison. The naive algorithm would shift P by just one place and begin comparing again from the left end of P. But a larger shift may often be possible. For example, if P $= abcxabcde$ and, in the present alignment of P with T, the mismatch occurs in position 8 of P, then it is easily deduced (and we will prove below) that P can be shifted by four places without passing over any occurrences of P in T. Notice that this can be deduced without even knowing what string T is or exactly how P is aligned with T. Only the location of the mismatch in P must be known. The Knuth-Morris-Pratt algorithm is based on this kind of reasoning to make larger shifts than the naive algorithm makes. We now formalize this idea.

[3] We will present several solutions to that set problem including the Aho–Corasick method in Section 3.4. For those reasons, and for its historical role in the field, we fully develop the Knuth-Morris-Pratt method here.

2.2.5. The good suffix rule in the search stage of Boyer–Moore

Having computed $L^r(i)$ and $l'(i)$ for each position i in P, these preprocessed values are used during the search stage of the algorithm to achieve larger shifts. If, during the search stage, a mismatch occurs at position $i - 1$ of P and $L'(i) > 0$, then the good suffix rule shifts P by $n - L'(i)$ places to the right, so that the $L^i(i)$-length prefix of the shifted P aligns with the $L^r(i)$-length suffix of the unshifted P, In the case that $L'(i) = 0$, the good suffix rule shifts P by $n - l'(i)$ places. When an occurrence of P is found, then the rule shifts P by $n - l'(2)$ places. Note that the rules work correctly even when $l^r(i) = 0$.

One special case remains. When the first comparison is a mismatch (i.e., $P(n)$ mismatches) then P should be shifted one place to the right.

2.2.6. The complete Boyer–Moore algorithm

We have argued that neither the good suffix rule nor the bad character rule shift P so far as to miss any occurrence of P. So the Boyer–Moore algorithm shifts by the largest amount given by either of the rules, We can now present the complete algorithm.

The Boyer–Moore algorithm

```
{Preprocessing stage)
    Given the pattern P,
    Compute L^i(i) and l'(i) for each position i of P,
    and compute R(x) for each character x ∈ Σ.
{Search stage)
    k := n;
    while k ≤ m do
        begin
        i := n;
        h := k;
        while i > 0 and P(i) = T(h) do
            begin
            i := i − 1;
            h := h − 1;
            end;
        if i = 0 then
            begin
            report an occurrence of P in T ending at position k.
            k := k + n − l'(2);
            end
        else
            shift P (increase k) by the maximum amount determined by the
            (extended) bad character rule and the good suffix rule.
        end;
```

Note that although we have always talked about "shifting P", and given rules to determine by how much P should be "shifted, there is no shifting in the actual implementation. Rather, the index k is increased to the point where the right end of P would be "shifted". Hence, each act of shifting P takes constant time.

We will later show, in Section 3.2, that by using the strong good suffix rule alone, the

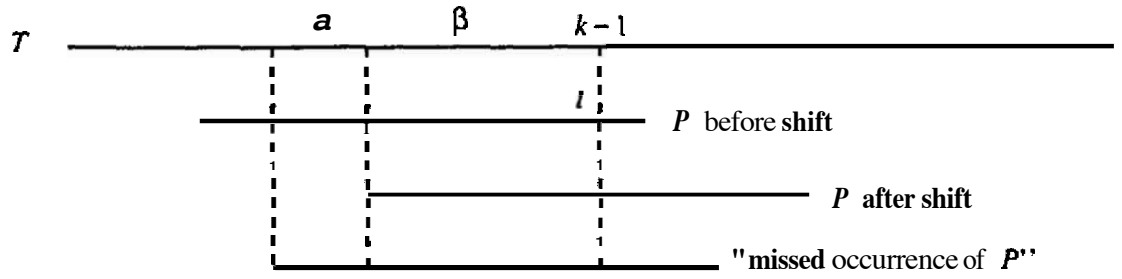

Figure 2.2: Assumed missed occurrence used in correctness proof for Knuth-Morris-Pratt.

by 4 places as shown below:

```
1         2
123456789012345678
xyabcxabcxadcdqfeg
  abcxabcde
     abcxabcde
```

As guaranteed, the first 3 characters of the shifted P match their counterparts in T (and their counterparts in the unshifted P).

Summarizing, we have

Theorem 2.3.1. *After a mismatch at position* i + 1 *of P and a shift of* i − sp'_i *places to the right, the left-most* sp'_i *characters of P are guaranteed to match their counterparts in T.*

Theorem 2.3.1 partially establishes the correctness of the Knuth-Morris-Pratt algorithm, but to fully prove correctness we have to show that the shift rule never shifts too far. That is, using the shift rule no occurrence of P will ever be overlooked.

Theorem 2.3.2. *For any alignment of P with T, if characters 1 through* i *of P match the opposing characters of T but character* i + 1 *mismatches* $T(k)$, *then P can be shifted by* i − sp'_i *places to the right without passing any occurrence of P in T.*

PROOF Suppose not, so that there is an occurrence of P starting strictly to the left of the shifted P (see Figure 2.2), and let a and β be the substrings shown in the figure. In particular, β is the prefix of P of length sp'_i, shown relative to the shifted position of P. The unshifted P matches T up through position i of P and position k − 1 of T, and all characters in the (assumed) missed occurrence of P match their counterparts in T. Both of these matched regions contain the substrings α and β, so the unshifted P and the assumed occurrence of P match on the entire substring $\alpha\beta$. Hence $\alpha\beta$ is a suffix of $P[1..i]$ that matches a proper prefix of P. Now let $l = |\alpha\beta| + 1$ so that position l in the "missed occurrence" of P is opposite position k in T. Character $P(l)$ cannot be equal to $P(i+1)$ since $P(l)$ is assumed to match $T(k)$ and $P(i+1)$ does not match $T(k)$. Thus $\alpha\beta$ is a proper suffix of $P[1..i]$ that matches a prefix of P, and the next character is unequal to $P(i+1)$. But $|\alpha| > 0$ due to the assumption that an occurrence of P starts strictly before the shifted P, so $|\alpha\beta| > |\beta| = sp'_i$, contradicting the definition of spf. Hence the theorem is proved. □

Theorem 2.3.2 says that the Knuth-Morris-Pratt shift rule does not miss any occurrence of P in T, and so the Knuth-Moms-Pratt algorithm will correctly find all occurrences of P in T. The time analysis is equally simple.

Definition For each position i in pattern P, define $sp_i(P)$ to be the *length* of the longest proper *suffix* of P[1..i] that matches a prefix of P.

Stated differently, $sp_i(P)$ is the length of the longest proper substring of $P[1..i]$ that ends at i and that matches a prefix of P. When the stting is clear by context we will use sp_i in place of the full notation.

For example, if $P = abcaeabcabd$, then $sp_2 = sp_3 = 0, sp_4 = 1, sp_8 = 3$, and $sp_{10} = 2$. Note that by definition, $sp_1 = 0$ for any string.

An optimized version of the Knuth-Morris-Pratt algorithm uses the following values.

Definition For each position i in pattern P, define $sp'_i(P)$ to be the length of the longest proper suffix of $P[1..i]$ that matches a prefix of P. *with the added condition that characters $P(i+1)$ and $P(sp'_i+1)$ are unequal.*

Clearly, $sp'_i(P) \leq sp_i(P)$ for all positions i and any string P. As an example, if P = bbccaebbcabd, then $sp_8 = 2$ because string bb occurs both as a proper prefix of $P[1..8]$ and as a suffix of $P[1..8]$. However, both copies of the string are followed by the same character c, and so $sp'_8 < 2$, In fact, $sp'_8 = 1$ since the single character b occurs as both the first and last character of $P[1..8]$ and is followed by character b in position 2 and by character c in position 9.

The Knuth-Morris-Pratt shift rule

We will describe the algorithm in terms of the sp' values, and leave it to the reader to modify the algorithm if only the weaker sp values are used.[4] The Knuth-Morris-Pratt algorithm aligns P with T and then compares the aligned characters from left to right, as the naive algorithm does.

For any alignment of P and T, if the first mismatch (comparing from left to sight) occurs in position $i+1$ of P and position k of T, then shift P to the right (relative to T) so that PII..sp'_i] aligns with T[k $- sp'_i..k - 1$]. En other words, shift P exactly $i + 1 - (sp'_i + 1) =$ i $- sp'_i$ places to the right, so that character $sp'_i + 1$ of P will align with character k of T. In the case that an occurrence of P has been found (no mismatch), shift P by $n - sp'_n$ places.

The shift rule guarantees that the prefix $P[1..sp'_i]$ of the shifted P matches its opposing substring in T. The next comparison is then made between characters $T(k)$ and $P[sp'_i + 1]$. The use of the stronger shift rule based on sp'_i guarantees that the same mismatch will not occur again in the new alignment, but it does not guarantee that $T(k) = P[sp'_i + 1]$.

In the above example, where $P = abcxabcde$ and $sp'_7 = 3$, if character 8 of P mismatches then P will be shifted by $7 - 3 = 4$ places. This is true even without knowing T or how P is positioned with T.

The advantage of the shift rule is twofold. First, it *often* shifts P by more than just a single character. Second, after a shift, the left-most sp'_i characters of P are guaranteed to match their counterparts in T. Thus, to determine whether the newly shifted P matches its counterpart in T, the algorithm *can start* comparing P and T at position $sp'_i + 1$ of P (and position k of T). For example, suppose P = abcxabcde as above, T = ***xyabcxabcxadcdqf*** eg, and the left end of P is aligned with character 3 of T. Then P and T will match for 7 characters but mismatch an character 8 of P, and P will be shifted

[4] The reader should be alerted that traditionally the Knuth-Morris-Pratt algorithm has been described in terms of *failure functions*, which are related to the sp_i values. Failure functions will be explicitly defined in Section 2.3.3.

$sp_n(P) := sp'_n(P)$;
for i := n − 1 downto 2 do
    $sp_i(P) := \max[sp_{i+1}(P) - 1, sp'_i(P)]$

2.3.3. A full implementation of Knuth-Morris-Pratt

We have described the Knuth-Morris-Pratt algorithm in terms of shifting P, but we never accounted for time needed to implement shifts. The reason is that shifting is only conceptual and P is never explicitly shifted. Rather, as in the case of Boyer–Moore, pointers to P and T are incremented. We use pointer p to point into P and one pointer c (for "current" character) to point into T.

Definition For each position i from 1 to n + 1, define the failure function $F'(i)$ to be $sp'_{i-1} + 1$ (and define F(i) = $sp_{i-1} + 1$), where sp'_0 and sp_0 are defined to be zero.

We will only use the (stronger) failure function $F'(i)$ in this discussion but will refer to $F(i)$ later,

After a mismatch in position i + 1 > 1 of P, the Knuth-Morris-Pratt algorithm "shifts" P so that the next comparison is between the character in position c of T and the character in position $sp'_i + 1$ of P. But $sp'_i + 1 = F'(i+1)$, so a general "shift" can be implemented in constant time by just setting p to $F'(i+1)$. Two special cases remain. When the mismatch occurs in position 1 of P, then p is set to $F'(1)$ = I and c is incremented by one. When an occurrence of P is found, then P is shifted right by $n - sp'_n$ places. This is implemented by setting $F'(n+1)$ to $sp'_n + 1$.

Putting all the pieces together gives the full Knuth-Morris-Pratt algorithm.

Knuth-Morris-Pratt algorithm

```
begin
Preprocess P to find F'(k) = sp'_{k-1} + 1 for k from 1 to n + 1.
    c := 1;
    p := I;
    While c + (n − p) ≤ m
    do begin
        While P(p) = T(c) and p ≤ n
        do begin
            p := p + 1;
            c := c + 1;
        end;
    if p = n + 1 then
        report an occurrence of P starting at position c − n of T.
    if p := 1 then c := c + 1
    p := F'(p);
    end;
end.
```

2.4. Real-time string matching

In the search stage of the Knuth-Morris-Pratt algorithm, P is aligned against a substring of T and the two strings are compared left to right until either all of P is exhausted (in which

Theorem 2.3.3. *In the Knuth-Morris-Pratt method, the number of character comparisons is at most 2m.*

PROOF Divide the algorithm into compare/shift phases, where a single phase consists of the comparisons done between successive shifts. After any shift, the comparisons in the phase go left to right and start either with the last character of T compared in the previous phase or with the character to its right. Since P is never shifted left, in any phase at most one comparison involves a character of T that was previously compared. Thus, the total number of character comparisons is bounded by $m + s$, where s is the number of shifts done in the algorithm. But $s < m$ since after m shifts the right end of P is certainly to the right of the right end of T, so the number of comparisons done is bounded by $2m$. □

2.3.2. Preprocessing for Knuth-Morris-Pratt

The key to the speed up of the Knuth-Morris-Pratt algorithm over the naive algorithm is the use of sp' (or sp) values. It is easy to see how to compute all the sp' and sp values from the Z values obtained during the fundamental preprocessing of P. We verify this below.

Definition Position j > 1 *maps to* i if $i = j + Z_j(P) - 1$. That is, j maps to i if i is the right end of a Z-box starting at j.

Theorem 2.3.4. *For any $i > 1$, $sp'_i(P) = Z_j = i - j + 1$, where $j > 1$ is the smallest position that maps to i. If there is no such j then $sp'_i(P) = 0$. For any $i > 1$, $sp_i(P) = i - j + 1$, where j is the smallestposition in the range $1 < j \leq i$ that maps to i or beyond. if there is no such j, then $sp_i(P) = 0$.*

PROOF If $sp'_i(P)$ is greater than zero, then there is a proper suffix α of $P[1..i]$ that matches a prefix of P, such that $P[i+1]$ does not match $P[|\alpha|+1]$. Therefore, letting j denote the start of α, $Z_j = |\alpha| = sp'_i(P)$ and j maps to i. Hence, if there is no j in the range $1 < j \leq i$ that maps to i, then $sp'_i(P)$ must be zero.

Now suppose $sp'_i(P) > 0$ and let j be as defined above. We claim that j is the smallest position in the range 2 to i that maps to i. Suppose not, and let j^* be a position in the range $1 < j^* < j$ that maps to i. Then $P[j^*..i]$ would be apropersuffixof $P[1..i]$ that matches a prefix (call it β) of P. Moreover, by the definition of mapping, $P(i+1) \neq P(|\beta|)$, so $sp'_i(P) \geq |\beta| > |\alpha|$, contradicting the assumption that $sp'_i = a$.

The proofs of the claims for $sp_i(P)$ are similar and are left as exercises. □

Given Theorem 2.3.4, all the sp' and sp values can be computed in linear time using the Z_i values as follows:

Z-based Knuth-Morris-Pratt

```
for i := 1 ton do
    sp'_i := 0;
for j := n downto 2 do
begin
    i := j + Z_j(P) - 1;
    sp'_i := Z_j;
end;
```

The sp values are obtained by adding the following:

shift rule, the method becomes real time because it still never reexamines a position in T involved in a match (a feature inherited from the Knuth-Morris-Pratt algorithm), and it now also never reexamines a position involved in a mismatch. So, the search stage of this algorithm never examines a character in T more than once. It follows that the search is done in real time. Below we show how to find all the $sp'_{(i,x)}$ values in linear time. Together, this gives an algorithm that does linear preprocessing of P and real-time search of T.

It is easy to establish that the algorithm finds all occurrences of P in T, and we leave that as an exercise.

2.4.2. Preprocessing for real-time string matching

Theorem 2.4.1. For P[i+1] # x, $sp'_{(i,x)}(P) = i - j + 1$, where j *is the* smallest ***position*** such that j maps to i and $P(Z_j + 1) = x$. *If there* is no ***such*** j then $sp'_{(i,x)}(P) = 0$.

The proof of this theorem is almost identical to the proof of Theorem 2.3.4 (page 26) and is left to the reader. Assuming (as usual) that the alphabet is finite, the following minor modification of the preprocessing given earlier for Knuth-Morris-Pratt (Section 2.3.2) yields the needed $sp'_{(i,x)}$ values in linear time:

Z-based real-time matching

```
for i := 1 to n do
    sp'(i,x) := 0 for every character x;
for j := n downto 2 do
    begin
        i := j + Z_j(P) - 1;
        x := P(Z_j + 1);
        sp'(i,x) := Z_j;
    end;
```

Note that the linear time (and space) bound for this method require that the alphabet Σ be finite. This allows us to do $|\Sigma|$ comparisons in constant time. If the size of the alphabet is explicitly included in the time and space bounds, then the preprocessing time and space needed for the algorithm is $O(|\Sigma|n)$.

2.5. Exercises

1. In "typical" applications of exact matching, such as when searching for an English word in a book, the simple bad character rule seems to be as effective as the extended bad character rule. Give a "hand-waving" explanation for this.
2. When searching for a single word or a small phrase in a large English text, brute force (the naive algorithm) is reported [184] to run faster than most other methods. Give a hand-waving explanation for this. In general terms, how would you expect this observation to hold up with smaller alphabets (say in DNA with an alphabet size of four), as the size of the pattern grows, and when the text has many long sections of similar but not exact substrings?
3. "Common sense" and the $\Theta(nm)$ worst-case time bound of the Boyer–Moore algorithm (using only the bad character rule) both would suggest that empirical running times increase with increasing pattern length (assuming a fixed text). But when searching in actual English

case an occurrence of P in T has been found) or until a mismatch occurs at some positions i+1 of P and k of T. In the latter case, if $sp'_i > 0$, then P is shifted right by $i - sp'_i$ positions, guaranteeing that the prefix $P[1..sp'_i]$ of the shifted pattern matches its opposing substring in T, No explicit comparison of those substrings is needed, and the next comparison is between characters $T(k)$ and $P(sp'_i + 1)$. Although the shift based on sp'_i guarantees that $P(i + 1)$ differs from $P(sp'_i + 1)$, it does not guarantee that $T(k) = P(sp'_i + 1)$. Hence $T(k)$ might be compared several times (perhaps $\Omega(|P|)$ times) with differing characters in P. For that reason, the Knuth-Morris-Pratt method is not a real-time method.

To be real time, a method must do at most a constant amount of work between the time it first examines any position in T and the time it last examines that position. In the Knuth-Morris-Pratt method, if a position of T is involved in a match, it is never examined again (this is easy to verify) but, as indicated above, this is not true when the position is involved in a mismatch. Note that the definition of real time only concerns the search stage of the algorithm. Preprocessing of P need not be real time. Note also that if the search stage is real time it certainly is also linear time.

The utility of areal-time matcher Is two fold. First, in certain applications, such as when the characters of the text are being sent to a small memory machine, one might need to guarantee that each character can be fully processed before the next one is due to arrive. If the processing time for each character is constant, independent of the length of the string, then such a guarantee may be possible. Second, in this particular real-time matcher, the shifts of P may be longer but never shorter than in the original Knuth-Morris-Pratt algorithm. Hence, the real-time matcher may run faster in certain problem instances.

Admittedly, arguments in favor of real-time matching algorithms over linear-time methods are somewhat tortured, and the real-time matching is more a theoretical issue than a practical one. Still, it seems worthwhile to spend a little timediscussing real-time matching.

2.4.1. Converting Knuth-Morris-Pratt to a real-time method

We will use the Z values obtained during fundamental preprocessing of P to convert the Knuth-Morris-Pratt method into a real-time method. The required preprocessing of P is quite similar to the preprocessing done in Section 2.3.2 for the Knuth-Morris-Pratt algorithm. For historical reasons, the resulting real-time method is generally referred to as a deterministic *finite-state* string matcher and is often represented with a finite state machine diagram. We will not use this terminology here and instead represent the method in pseudo code.

Definition Let x denote a character of the alphabet. For each position i in pattern P, define $sp'_{(i,x)}(P)$ to be the length of the longest proper suffix of $P[1..i]$ that matches a prefix of P, *with the* added condition thnr *character* $P(sp'_i + 1)$ is x.

Knowing the $sp'_{(i,x)}$ values for each character x in the alphabet allows a shift rule that converts the Knuth-Morris-Pratt method into a real-time algorithm. Suppose P is compared against a substring of T and a mismatch occurs at characters $T(k) =$ x and P(i+1). Then P should be shifted right by $i - sp'_{(i,x)}$ places. This shift guarantees that the prefix $P[1..sp'_{(i,x)}]$ matches the opposing substring in T and that $T(k)$ matches the next character in P. Hence, the comparison between $T(k)$ and $P(sp'_{(i,x)} + 1)$ can be skipped. The next needed comparison is between characters $P(sp'_{(i,x)} + 2)$ and $T(k + 1)$. With this

Using *sp* values to compute Zvalues

In Section 2.3.2, we showed that one can compute all the *sp* values knowing only the Z values for string S (i.e., not knowing S itself). In the next five exercises we establish the converse, creating a linear-time algorithm to compute all the Z values from *sp* values alone. The first exercise suggests a natural method to accomplish this, and the following exercise exposes a hole in that method. The final three exercises develop a correct linear-time algorithm, detailed in [202]. We say that sp_i *maps* to k if $k = i - sp_i + 1$,

16. Suppose there is a position i such that sp_i maps to k, and let i be the largest such position. Prove that $Z_k = i - k + 1 = sp_i$ and that $r_k = i$.

17. Given the answer to the previous exercise, it is natural to conjecture that Z_k always equals sp_i, where i is the largest position such that sp_i maps to k. Show that this is not true. Given an example using at least three distinct characters.

Stated another way, give an example to show that Z_k can be greater than zero even when there is no position i such that sp_i maps to k.

18. Recall that r_{k-1} is known at the start of iteration k of the **Z** algorithm (when Z_k is computed), but r_k is known only at the end of iteration k. Suppose, however, that r_k is known (somehow) at the start of iteration k. Show how the Z algorithm can then be modified to compute Z_k using no character comparisons. Hence this modified algorithm need not even know the string S.

19. Prove that if Z_k is greater than zero, then r_k equals the largest position i such that $k \geq i - sp_i$. Conclude that r_k can be deduced from the *sp* values for every position k where Z_k is not zero.

20. Combine the answers to the previous two exercises to create a linear-time algorithm that computes all the **Z** values for a string S given only the sp values for S and not the string S itself.

Explain in what way the method is a "simulation" of the **Z** algorithm.

21. It may seem that $l'(i)$ (needed for Boyer–Moore) should be sp_n for any i. Show why this is not true.

22. In Section **1.5** we showed that all the occurrences of P in T could be found in linear time by computing the Z values on the string S = P\$T. Explain how the method would change if we use S = PT, that is, we do not use a separator symbol between P and T. Now show how to find all occurrences of P in T in linear time using S = PT, but with *sp* values in place of Z values. (This is not as simple as it might at first appear.)

23. In Boyer–Moore and Boyer–Moore–like algorithms, the search moves right to left in the pattern, although the pattern moves left to right relative to the text. That makes it more difficult to explain the methods and to combine the preprocessing for Boyer–Moore with the preprocessing for Knuth-Morris-Pratt. However, a small change to Boyer–Moore would allow an easier exposition and more uniform preprocessing. First, place the pattern at the *right* end of the text, and conduct each search *left* to *right* in the pattern, shifting the pattern *left* after a mismatch. Work out the details of this approach, and show how it allows a more uniform exposition of the preprocessing needed for it and for Knuth-Morris-Pratt. Argue that on average this approach has the same behavior as the original Boyer–Moore method.

24. Below is working Pascal code (in Turbo Pascal) implementing Richard Cole's preprocessing, for the strong good suffix rule. It is different than the approach based on fundamental preprocessing and is closer to the original method in [278]. Examine the code to extract the algorithm behind the program. Then explain the idea of the algorithm, prove correctness of the algorithm, and analyze its running time. The point of the exercise is that it is difficult to convey an algorithmic idea using a program.

texts, the Boyer–Moore algorithm runs faster in practice when given longer patterns. Thus, on an English text of about 300,000 characters, it took about five times as long to search for the word "Inter" as it did to search for "Interactively".

Give a hand-waving explanation for this. Consider now the case that the pattern length increases without bound. At what point would you expect the search times to stop decreasing? Would you expect search times to start increasing at some point?

4. Evaluate empirically the utility of the extended bad character rule compared to the original bad character rule. Perform the evaluation in combination with different choices for the two good-suffix rules. How much more is the average shift using the extended rule? Does the extra shift pay for the extra computation needed to implement it?

5. Evaluate empirically, using different assumptions about the sizes of P and T, the number of occurrences of P in T, and the size of the alphabet, the following idea for speeding up the Boyer–Moore method. Suppose that a phase ends with a mismatch and that the good suffix rule shifts Pfarther than the extended bad character rule. Let x and y denote the mismatching characters in T and P respectively, and let z denote the character in the shifted Pbelow x. By the suffix rule, z will not be y, but there is no guarantee that it will be x. So rather than starting comparisons from the right of the shifted P, as the Boyer–Moore method would do, why not first compare x and z? If they are equal then a right-to-left comparison is begun from the right end of P, but if they are unequal then we apply the extended bad character rule from z in P. This will shift P again. At that point we must begin a right-to-left comparison of P against T.

6. The idea of the bad character rule in the Boyer–Moore algorithm can be generalized so that instead of examining characters in P from right to left, the algorithm compares characters in P in the order of how unlikely they are to be in T (most *unlikely* first). That is, it looks first at those characters in P that are least likely to be in T. Upon mismatching, the bad character rule or extended bad character rule is used as before. Evaluate the utility of this approach, either empirically on real data or by analysis assuming random strings.

7. Construct an example where fewer comparisons are made when the bad character rule is used alone, instead of combining it with the good suffix rule.

8. Evaluate empirically the effectiveness of the strong good suffix shift for Boyer–Moore versus the weak shift rule.

9. Give a proof of Theorem 2.2.4. Then show how to accumulate all the $l'(i)$ values in linear time.

10. If we use the weak good suffix rule in Boyer–Moore that shifts the closest copy of t under the matched suffix t, but doesn't require the next character to be different, then the preprocessing for Boyer–Moore can be based directly on sp, values rather than on Z values. Explain this.

11. Prove that the Knuth-Morris-Pratt shift rules (either based on sp or sp') do not miss any occurrences of P in T.

12. It is possible to incorporate the bad character shift rule from the Boyer–Moore method to the Knuth-Morris-Pratt method or to the naive matching method itself. Show how to do that. Then evaluate how effective that rule is and explain why it is more effective when used in the Boyer–Moore algorithm.

13. Recall the definition of l_i on page 8. It is natural to conjecture that $sp_i = i - l_i$ for any index i, where $i \geq l_i$. Show by example that this conjecture is incorrect.

14. Prove the claims in Theorem 2.3.4 concerning $sp'_i(P)$.

15. Is it true that given only the sp values for a given string P, the sp' values are completely determined? Are the sp values determined from the sp' values alone?

```
        if (p[k] = p[j]) then
           begin I3}
           kmp_shift[k]:=j-k;
          j:=j-1;
           end I31
        else
        kmp_shift[k]:=j-k+1;
     end; {2}

  {stage 21
   j:=j+1;
   j_old:=1;

   while (j <= m) do
     begin {2}
       for i:=j_old to j-1 do
       if (gs_shift[i] > j-1) then gs_shift[i]:=j-1;

       j_old:=j;
       j:=j+kmp_shift[j];
     end; {2}
  end: {1}

begin {main}

writeln('input a string on a single line');

readstring(p,m);
gsshift(p,matchshift,m);
writeln('the value in cell i is the number of positions to shift');
writeln('after a mismatch occurring in position i of the pattern');

for i:= 1 to m do
write(matchshift[i]:3);
writeln;
end. {main}
```

25. Prove that the shift rule used by the real-time string matcher does not miss any occurrences of P in T.

26. Prove Theorem 2.4.1.

27. In this chapter, we showed how to use Z values to compute both the sp'_i and sp_i values used in Knuth-Morris-Pratt and the $sp'_{i,x}$ values needed for its real-time extension. instead of using Z values for the $sp'_{i,x}$ values, show how to obtain these values from the sp_i and/or sp'_i values in linear $[O(n|\Sigma|)]$ time, where n is the length of P and $|\Sigma|$ is the length of the alphabet.

28. Although we don't know how to simply convert the Boyer–Moore algorithm to be a real-time method the way Knuth-Morris-Pratt was converted, we can make similar changes to the strong shift rule to make the Boyer–Moore shift more effective. That is, when a mismatch occurs between $P(i)$ and $T(h)$ we can look for the right-most copy in P of $P[i+1..n]$ (other than $P[i+1..n]$ itself) such that the preceding character is $T(h)$. Show how to modify

```
program gsmatch(input,output);
(This  is an implementation of Richard Cole's
preprocessing for the strong good suffix rule}
type
tstring = string[200];
indexarray = array[1..100] of integer;

const
zero = 0;

var
p:tstring;
bmshift,matchshift:indexarray;
m,i:integer;

procedure readstring(var p:tstring; var m:integer);

begin
read(p);

m:=length(p);
writeln('the length of the string is ', m);

end;

procedure gsshift(p:tstring; var
gs_shift:indexarray;m:integer);

var
i,j,j_old,k:integer;
kmp_shift:indexarray;
go-on:boolean;

begin {1}
  for j:= 1 to m do
    gs_shift[j] := m;
  kmp_shift[m]:=1;

{stage 1}
  j:=m;

  for k:=m-1 downto 1 do
    begin (2 1
       go_on:=true;
       while (p[j] <> p[k]) and go-on do
          begin I31
          if (gs_shift[j] > j-k) then gs_shift[j] := j-k;
          if (j < m) then j:= j+kmp_shift[j+1]
          else go_on:=false;
          end; {3}
```

3

Exact Matching: A Deeper Look at Classical Methods

3.1. A Boyer–Moore variant with a "simple" linear time bound

Apostolico and Giancarlo [26] suggested a variant of the Boyer–Moore algorithm that allows a fairly simple proof of linear worst-case running time. With this variant, no character of T will ever be compared after it is first matched with any character of P. It is then immediate that the number of comparisons is at most 2m: Every comparison is either a match or a mismatch; there can only be m mismatches since each one results in a nonzero shift of P; and there can only be m matches since no character of T is compared again after it matches a character of P. We will also show that (in addition to the time for comparisons) the time taken for all the other work in this method is linear in m.

Given the history of very difficult and partial analyses of the Boyer–Moore algorithm, it is quite amazing that a close variant of the algorithm allows a simple linear time bound. We present here a further improvement of the Apostolico-Giancarlo idea, resulting in an algorithm that simulates ***exactly*** the shifts of the Boyer–Moore algorithm. The method therefore has all the rapid shifting advantages of the Boyer–Moore method as well as a simple linear worst-case time analysis.

3.1.1. Key ideas

Our version of the Apostolico-Giancarlo algorithm simulates the Boyer–Moore algorithm, finding exactly the same mismatches that Boyer–Moore would find and making exactly the same shifts. However, it infers and avoids many of the explicit matches that Boyer–Moore makes.

We take the following high-level view of the Boyer–Moore algorithm. We divide the algorithm into *compare/shift phases* numbered 1 through q $\leq$ m. In a compare/shift phase, the right end of P is aligned with a character of T, and P is compared right to left with selected characters of T until either all of P is matched or until a mismatch occurs. Then, P is shifted right by some amount as given in the Boyer–Moore shift rules.

Recall from Section 2.2.4, where preprocessing for Boyer–Moore was discussed, that $N_i(P)$ is the length of the longest suffix of $P[1..i]$ that matches a suffix of P. In Section 2.2.4 we showed how to compute N_i for every i in $O(n)$ time, where n is the length of P. We assume here that vector N has been obtained during the preprocessing of P.

Two modifications of the Boyer–Moore algorithm are required. First, during the search for P in T (after the preprocessing), we maintain an m length vector M in which at most one entry is updated in every phase. Consider a phase where the right end of P is aligned with position j of T and suppose that P and T match for l places (from right to left) but no farther. Then, set $M(j)$ to a value k $\leq l$ (the rules for selecting k are detailed below). $M(j)$ records the fact that a suffix of P of length k (at least) occurs in T and ends exactly

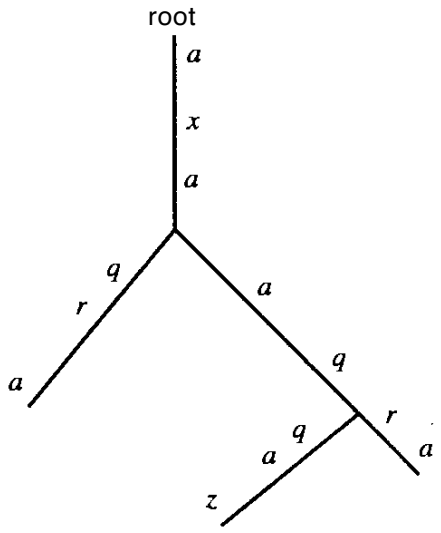

Figure 2.3: The pattern $P = aqra$ labels two subpaths of paths starting at the root. Those paths start at the root, but the subpaths containing *aqra* do not. There is also another subpath in the tree labeled *aqra* (it starts above the character *z*), but it violates the requirement that it be a subpath of a path starting at the root, Note that an edge label is displayed from the top of the edge down towards the bottom of the edge. Thus in the figure, there is an edge labeled "qra", not "arq".

the Boyer–Moore preprocessing so that the needed information is collected in linear time, assuming a fixed size alphabet.

29. Suppose we are given a tree where each edge is labeled with one or more characters, and we are given a pattern P. The label of a subpath in the tree is the concatenation of the labels on the edges in the subpath. The problem is to find all subpaths of paths starting at the root that are labeled with pattern P. Note that although the subpath must be part of a path directed from the root, the subpath itself need not start at the root (see Figure 2.3). Give an algorithm for this problem that runs in time proportional to the total number of characters on the edges of the tree plus the length of P.

4. If $M(h) > N_i$ and $N_i < i$, then P matches T from the right end of P down to character $i - N_i + 1$ of P, but the next pair of characters mismatch [i.e., $P(i - N_i) \neq T(h - N_i)$]. Hence P matches T for $j - h + N_i$ characters and mismatches at position $i - N_i$ of P. $M(j)$ must be set to a value less than or equal to $j - h + N_i$. Set $M(j)$ to $j - h$. Shift P by the Boyer–Moore rules based on a mismatch at position $i - N_i$ of P (this ends the phase).
5. If $M(h) = N_i$ and $0 < N_i < i$, then P and T must match for at least $M(h)$ characters to the left, but the left end of P has not yet been reached, so set i to $i - M(h)$ and set h to $h - M(h)$ and repeat the phase algorithm.

3.1.3. Correctness and linear-time analysis

Theorem 3.1.1. *Using* M *and* N *as above, the Apostolico–Giancarlo variant of the Boyer–Moore algorithm correctly finds all occurrences of P in T.*

PROOF We prove correctness by showing that the algorithm simulates the original Boyer–Moore algorithm. That is, for any given alignment of P with T, the algorithm is correct when it declares a match down to a given position and a mismatch at the next position. The rest of the simulation is correct since the shift rules are the same as in the Boyer–Moore algorithm.

Assume inductively that $M(h)$ values are valid up to some position in T. That is, wherever $M(h)$ is defined, there is an $M(h)$-length substring in T ending at position h in T that matches a suffix of P. The first such value, $M(n)$, is valid because it is found by aligning P at the left of T and making explicit comparisons, repeating rule 1 of the phase algorithm until a mismatch occurs or an occurrence of P is found. Now consider a phase where the right end of P is aligned with position j of T. The phase simulates the workings of Boyer–Moore except that in cases 2, 3, 4, and 5 certain explicit comparisons are skipped and in case 4 a mismatch is inferred, rather than observed. But whenever comparisons are skipped, they are certain to be matches by the definition of N and M and the assumption that the M values are valid thus far. Thus it is correct to skip these comparisons. In case 4, a mismatch at position i – N, of P is correctly inferred because N_i is the maximum length of any substring ending at i that matches a suffix of P, whereas $M(h)$ is less than or equal to the maximum length of any substring ending at h that matches a suffix of P. Hence this phase correctly simulates Boyer–Moore and finds exactly the same mismatch (or an occurrence of P in T) that Boyer–Moore would find. The value given to $M(j)$ is valid since in all cases it is less than or equal to the length of the suffix of P shown to match its counterpart in the substring $T[1..i]$. □

The following definitions and lemma will be helpful in bounding the work done by the algorithm.

Definition If j is a position where $M(j)$ is greater than zero then the interval $[j - M(j) + 1..j]$ is called a ***covered interval*** defined by j.

Definition Let $j' < j$ and suppose covered intervals are defined for both j and j'. We say that the covered intervals for j and j' ***cross*** if $j - M(j) + 1 \leq j'$ and $j' - M(j') + 1 < j - M(j) + 1$ (see Figure 3.2).

Lemma 3.1.1. *No covered intervals computed by the algorithm ever cross each other. Moreover, if the algorithm examines a position h of T in a covered interval, then h is at the right end of that interval.*

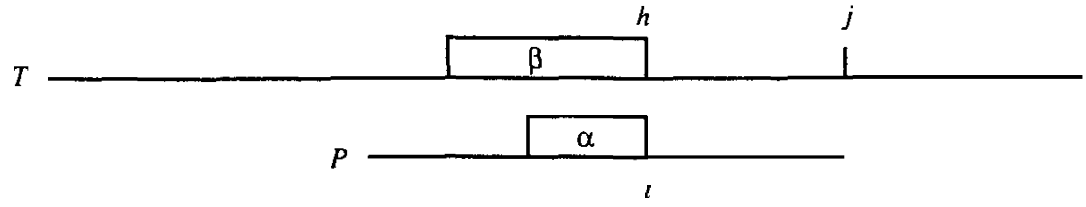

Figure 3.1: Substring α has length N_i and substring β has length $M(h) > N_i$. The two strings must match from their right ends for N_i characters, but mismatch at the next character.

at position j. As the algorithm proceeds, a value for $M(j)$ is set for every position j in T that is aligned with the right end of P; $M(j)$ is undefined for all other positions in T.

The second modification exploits the vectors N and M to speed up the Boyer–Moore algorithm by inferring certain matches and mismatches. To get the idea, suppose the Boyer–Moore algorithm is about to compare characters $P(i)$ and $T(h)$, and suppose it knows that $M(h) > N_i$ (see Figure 3.1). That means that an N_i-length substring of P ends at position i and matches a suffix of P, while an $M(h)$-length substring of T ends at position h and matches a suffix of P. So the N_i-length suffixes of those two substrings must match, and we can conclude that the next N_i comparisons (from $P(i)$ and $T(h)$ moving leftward) in the Boyer–Moore algorithm would be matches. Further, if $N_i = i$, then an occurrence of P in T has been found, and if $N_i < i$, then we can be sure that the next comparison (after the N_i matches) would be a mismatch. Hence in simulating Boyer–Moore, if $M(h) > N_i$ we can avoid at least N_i explicit comparisons. Of course, it is not always the case that $M(h) > N_i$, but all the cases are similar and are detailed below.

3.1.2. One phase in detail

As in the original Boyer–Moore algorithm, when the right end of P is aligned with a position j in T, P is compared with T from right to left. When a mismatch is found or inferred, or when an occurrence of P is found, P is shifted according to the original Boyer–Moore shift rules (either the strong or weak version) and the compare/shift phase ends. Here we will only give the details for a single phase. The phase begins with h set to j and i set to n.

Phase algorithm

1. If $M(h)$ is undefined or $M(h) = N_i = 0$, then compare $T(h)$ and $P(i)$ as follows:

   If $T(h) = P(i)$ and i = 1, then report an occurrence of P ending at position j of T, set $M(j) =$ n, and shift as in the Boyer–Moore algorithm (ending this phase).

   If $T(h) = P(i)$ and i > 1, then set h to h − 1 and i to i − 1 and repeat the phase algorithm.

   If $T(h) \neq P(i)$, then set $M(j) = j - h$ and shift P according to the Boyer–Moore rules based on a mismatch occurring in position i of P (this ends the phase).

2. If $M(h) < N_i$, then P matches its counterparts in T from position n down to position i − $M(h)$ + 1 of P. By the definition of $M(h)$, P might match more of T to the left, so set i to $i - M(h)$, set h to h − $M(h)$, and repeat the phase algorithm.

3. If $M(h) \geq N_i$ and $N_i = i > 0$, then declare that an occurrence of P has been found in T ending at position j. $M(j)$ must be set to a value less than or equal to n. Set $M(j)$ to j − h, and shift according to the Boyer–Moore rules based on finding an occurrence of P ending at j (this ends the phase).

if the comparison involving $T(h)$ is a match then, at the end of the phase, $M(j)$ is set at least as large as $j - h + 1$. That means that all characters in T that matched a character of P during that phase are contained in the covered interval $[j - M(j) + 1..j]$. Now the algorithm only examines the right end of an interval, and if h is the right end of an interval then $M(h)$ is defined and greater than 0, so the algorithm never compares a character of T in a covered interval. Consequently, no character of T will ever be compared again after it is first in a match. Hence the algorithm finds at most m matches, and the total number of character comparisons is bounded by $2m$.

To bound the amount of additional work, we focus on the number of accesses of M during execution of the five cases since the amount of additional work is proportional to the number of such accesses. A character comparison is done whenever Case 1 applies. Whenever Case 3 or 4 applies, P is immediately shifted. Hence Cases 1, 3, and 4 can apply at most $O(m)$ times since there are at most $O(m)$ shifts and compares. However, it is possible that Case 2 or Case 5 can apply without an immediate shift or immediate character comparison. That is, Case 2 or 5 could apply repeatedly before a comparison or shift is done. For example, Case 5 would apply twice in a row (without a shift or character comparison) if $N_i = M(h) > 0$ and $N_{i-N_i} = M(h - M(h))$. But whenever Case 2 or 5 applies, then $j > h$ and $M(j)$ will certainly get set to $j - h + 1$ or more at the end of that phase. So position h will be in the strict interior of the covered interval defined by j. Therefore, h will never be examined again, and $M(h)$ will never be accessed again. The effect is that Cases 2 and 5 can apply at most once for any position in T, so the number of accesses made when these cases apply is also $O(m)$. □

3.2. Cole's linear worst-case bound for Boyer–Moore

Here we finally present a linear worst-case time analysis of the *original* Boyer–Moore algorithm. We consider first the use of the (strong) good suffix rule by itself. Later we will show how the analysis is affected by the addition of the bad character rule. Recall that the good suffix rule is the following:

> Suppose for a given alignment of P and T, a substring t of T matches a suffix of P, but a mismatch occurs at the next comparison to the left. Then find, if it exists, the right-most copy t' of t in P such that t' is not a suffix of P and *such that the character to the left of* t' *differs from the mismatched character* in P. Shift P to the right so that substring t' in P is below substring t in T (recall Figure 2.1). If t' does not exist, then shift the left end of P past the left end of t in T by the least amount so that a prefix of the shifted pattern matches a suffix of t in T. If no such shift is possible, then shift P by n places to the right.
>
> If an occurrence of P is found, then shift P by the least amount so that a *proper* prefix of the shifted pattern matches a suffix of the occurrence of P in T. If no such shift is possible, then shift P by n places.

We will show that by using the good suffix rule alone, the Boyer–Moore method has a worst-case running time of $O(m)$, provided that the pattern does not appear in the text. Later we will extend the Boyer–Moore method to take care of the case that P does occur in T.

As in our analysis of the Apostolico–Giancarlo algorithm, we divide the Boyer–Moore algorithm into *compare/shift* phases numbered 1 through q $\leq m$. In compare/shift phase i, a suffix of P is matched right to left with characters of T until either all of P is matched or until a mismatch occurs. In the latter case, the substring of T consisting of the matched

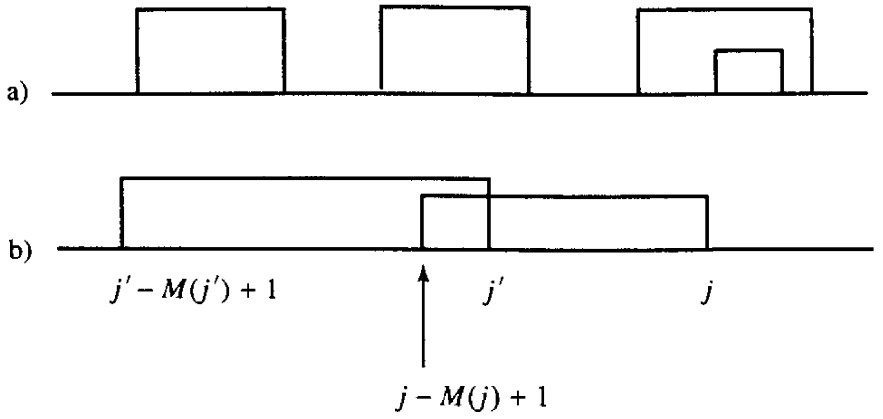

Figure 3.2: a. Diagram showing covered intervals that do not cross, although one interval can contain another. b. Two covered intervals that do cross.

PROOF The proof is by induction on the number of intervals created. Certainly the claim is true until the first interval is created, and that interval does not cross itself. Now assume that no intervals cross and consider the phase where the right end of P is aligned with position j of T.

Since $h = j$ at the start of the phase, and j is to the right of any interval, h begins outside any interval. We consider how h could first be set to a position inside an interval, other than the right end of the interval. Rule 1 is never executed when h is at the right end of an interval (since then $M(h)$ is defined and greater than zero), and after any execution of Rule 1, either the phase ends or h is decremented by one place. So an execution of Case 1 cannot cause h to move beyond the right-most character of a covered interval. This is also true for Cases 3 and 4 since the phase ends after either of those cases. So if h is ever moved into an interval in a position other than its right end, that move must follow an execution of Case 2 or 5. An execution of Case 2 or 5 moves h from the right end of some interval $I = [k..h]$ to position $k - 1$, one place to the left of I. Now suppose that $k - 1$ is in some interval I' but is not at its right end, and that this is the first time in the phase that h (presently $k - 1$) is in an interval in a position other than its right end. That means that the right end of I cannot be to the left of the right end of I' (for then position $k - 1$ would have been strictly inside I'), and the right ends of I and I' cannot be equal (since $M(h)$ has at most one value for any h). But these conditions imply that I and I' cross, which is assumed to be untrue. Hence, if no intervals cross at the start of the phase, then in that phase only the right end of any covered interval is examined.

A new covered interval gets created in the phase only after the execution of Case 1, 3, or 4. In any of these cases, the interval $[h + 1..j]$ is created after the algorithm examines position h. In Case 1, h is not in any interval, and in Cases 3 and 4, h is the right end of an interval, so in all cases $h + 1$ is either not in a covered interval or is at the left end of an interval. Since j is to the right of any interval, and $h + 1$ is either not in an interval or is the left end of one, the new interval $[h + 1..j]$ does not cross any existing interval. The previously existing intervals have not changed, so there are no crossing intervals at the end of the phase, and the induction is complete. □

Theorem 3.1.2. ***The modified Apostolico–Giancarlo algorithm does at most 2m character comparisons and at most $O(m)$ additional work.***

PROOF Every phase ends if a comparison finds a mismatch and every phase, except the last, is followed by a nonzero shift of P. Thus the algorithm can find at most m mismatches. To bound the matches, observe that characters are explicitly compared only in Case 1, and

be periodic. For example, *abababab* is periodic with period *abab* and also with shorter period *ab*. An alternate definition of a semiperiodic string is sometimes useful.

Definition A string α is *prefix semiperiodic* with period γ if α consists of one or more copies of string γ followed by a nonempty prefix (possibly the entire γ) of string γ.

We use the term "prefix semiperiodic" to distinguish this definition from the definition given for "semiperiodic", but the following lemma (whose proof is simple and is left as an exercise) shows that these two definitions are really alternate reflections of the same structure.

Lemma 3.2.2. *A string α is semiperiodic with period β if and only if it is prefix semiperiodic with the same length period as β.*

For example, the string *abaabaabaabaabaah* is serniperiodic with period *aab* and is prefix semiperiodic with period *aba*.

The following useful lemma is easy to verify, and its proof is typical of the style of thinking used in dealing with overlapping matches.

Lemma 3.2.3. *Suppose pattern P occurs in text T starting at positions p and $p' > p$, where $p' - p \leq \lfloor n/2 \rfloor$. Then P is serniperiodic with period $p' - p$.*

The following lemma, called the GCD *Lemma,* is a very powerful statement about periods of strings. We won't need the lemma in our discussion of Cole's proof, but it is natural to state it here. We will prove it and use it in Section 16.17.5.

Lemma 3.2.4. *Suppose string α is semiperiodic with both a period of length p and a period of length q, and $|\alpha| \geq p + q$. Then α is semiperiodic with a period whose length is the greatest common divisor of p and q.*

Return to Cole's proof

Recall that the key thing to prove is that $s_i \geq g_i/3$ in every phase i. As noted earlier, it then follows easily that the total number of comparisons is bounded by $4m$.

Consider the ith compare/shift phase, where substring t_i of T matches a suffix of P and then P is shifted s_i places to the right. If $s_i \geq (|t_i| + 1)/3$, then $s_i \geq g_i/3$ even if all characters of T that were compared in phase i had been previously compared. Therefore, it is easy to handle phases where the shift is "relatively" large compared to the total number of characters examined during the phase. Accordingly, for the next several lemmas we consider the case when the shift is relatively small (i.e., $s_i < (|t_i| + 1)/3$ or, equivalently, $|t_i| + 1 > 3s_i$).

We need some notation at this point. Let a be the suffix of P of length s_i, and let β be the smallest substring such that $\mathrm{a} = \beta^l$ for some integer l (it may be that $\beta = \mathrm{a}$ and $l = 1$). Let $\bar{P} = P[n - |t_i|..n]$ be the suffix of P of length $|t_i| + l$, that is, that portion of P (including the mismatch) that was examined in phase i. See Figure *3.3*.

Lemma 3.2.5. **If** *$|t_i| + l > 3s_i$, then both t_i and $\bar{P}$ are semiperiodic with period a and hence with period β.*

PROOF Starting from the right end of $\bar{P}$, mark off substrings of length s_i until less than s_i characters remain on the left (see Figure 3.4). There will be at least three full substrings since $|\bar{P}| = |t_i| + l > 3s_i$. Phase i ends by shifting P right by s_i positions. Consider how $\bar{P}$ aligns with T before and after that shift (see Figure 3.5). By definition of s_i and a , a is the part of the shifted $\bar{P}$ to the right of the original $\bar{P}$. By the good suffix rule, the portion

characters is denoted t_i, and the mismatch occurs just to the left of t_i. The pattern is then shifted right by an amount determined by the good suffix rule.

3.2.1. Cole's proof when the pattern does not occur in the text

Definition Let s_i denote the amount by which P is shifted right at the end of phase i.

Assume that P does not occur in T, so the compare part of every phase ends with a mismatch. In each compare/shift phase, we divide the comparisons into those that compare a character of T that has previously been compared (in a previous phase) and those comparisons that compare a character of T for the first time in the execution of the algorithm. Let g_i be the number of comparisons in phase i of the first type (comparisons involving a previously examined character of T), and let g_i' be the number of comparisons in phase i of the second type. Then, over the entire algorithm the number of comparisons is $\sum_{i=1}^{q}(g_i + g_i')$, and our goal is to show that this sum is $O(m)$.

Certainly, $\sum_{i=1}^{q} g_i' \leq m$ since a character can be compared for the first time only once. We will show that for any phase i, s, $\geq g_i/3$. Then since $\sum_{i=1}^{q} s_i \leq$ m (because the total length of all the shifts is at most rn) it will follow that $\sum_{i=1}^{q} g_i \leq 3m$. Hence the total number of comparisons done by the algorithm is $\sum_{i=1}^{q}(g_i + g_i') \leq 4$m.

An initial lemma

We start with the following definition and a lemma that is valuable in its own right.

Definition For any string β, β^i denotes the string obtained by concatenating together i copies of β.

Lemma 3.2.1. Let y and S be two nonernpty strings such that $\gamma\delta = 6$y. Then $6 = p^i$ and $y = \rho^j$ for some string p and positive integers i and j.

This lemma says that if a string is the same before and after a circular shift (so that it can be written both as $\gamma\delta$ and Sy, for some strings y and 6) then γ and 6 can both be written as concatenations of some single string p.

For example, let $\delta =$ ababa and $\gamma = ababab$, so $\delta\gamma =$ abnbabababab $= \gamma\delta$. Then $\rho =$ ab, $6 = \rho^2$, and $y = p^3$.

PROOF The proof is by induction on $|\delta| + |y|$. For the basis, if $|\delta| + |\gamma| = 2$, it must be that $6 = y = \rho$ and $i = j = 1$. Now consider larger lengths. If $|\delta| = |\gamma|$, then again $\delta = y =$ p and $i =$ j $=$ I. So suppose $|\delta| < |\gamma|$. Since $\delta\gamma = \gamma\delta$ and $|\delta| < |\gamma|$, 6 must be a prefix of y, so $y = 66'$ for some string 6'. Substituting this into $\delta\gamma = \gamma\delta$ gives $666' = \delta\delta'\delta$. Deleting the left copy of 6 from both sides gives $\delta\delta' = \delta'\delta$. However, $|\delta| + |\delta'| = |\gamma| < |\delta| + |\gamma|$, and so by induction, $6 = p^i$ and $\delta' = \rho^j$. Thus, $y = SS' = \rho^k$, where $k = i + j$. □

Definition A string a is *semiperiodic* with period β if a consists of a nonempty suffix of a string β (possibly the entire β) followed by one or more copies of β. String a is called periodic withperiod β if a consists of two or more complete copies of β. We say that string α is *periodic* if it is periodic with some period β.

For example, bcabcabc is semiperiodic with period abc, but it is not periodic. String abcabc is periodic with period abc. Note that a periodic string is by definition also semiperiodic. Note also that a string cannot have itself as a period although a period may itself

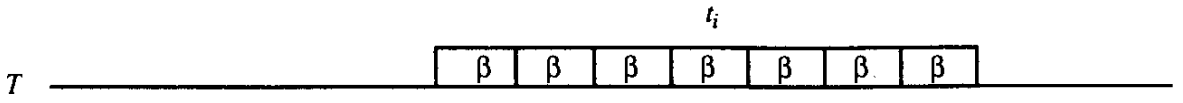

Figure 3.6: Substring t, in T is semiperiodic with period β.

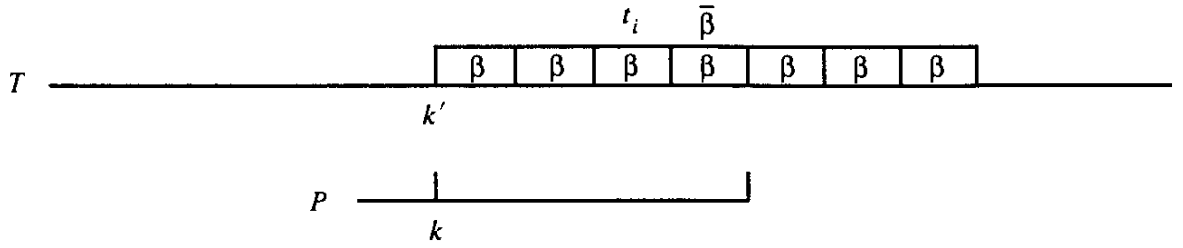

Figure 3.7: The case when the right end of P is aligned with a right end of $\bar{\beta}$ in phase h. Here $q = 3$. A mismatch must occur between $T(k')$ and $P(k)$.

concreteness, call that copy $\bar{\beta}$ and say that its right end is $q|\beta|$ places to the left of the right of t_i, where $q \geq 1$ (see Figure **3.7**). We will first deduce how phase h must have ended, and then we'll use that to prove the lemma.

Let k' be the position in T just to the left of t_i (so $T(k')$ is involved in the mismatch ending phase i), and let k be the position in P opposite $T(k')$ in phase h. We claim that, in phase h, the comparison of P and T will find matches until the left end of t, but then mismatch when comparing $T(k')$ and $P(k)$. The reason is the following: Strings $\bar{P}$ and t_i are semiperiodic with period β, and in phase h the right end of P is aligned with the right end of some β. So in phase h, P and T will certainly match until the left end of string t_i. Now $\bar{P}$ is semiperiodic with β, and in phase h, the right end of P is exactly $q|\beta|$ places to the left of the right end of t_i. Therefore, $\bar{P}(1) = \bar{P}(1 + |\beta|) = \cdots = \bar{P}(1 + q|\beta|) = P(k)$. But in phase i the mismatch occurs when comparing $T(k')$ with $\bar{P}(1)$, so $P(k) = \bar{P}(1) \neq$ T(k'). Hence, if in phase h the right end of P is aligned with the right end of a β, then phase h must have ended with a mismatch between T(k') and $P(k)$. This fact will be used below to prove the lemma.[1]

Now we consider the possible shifts of P done in phase h. We will show that every possible shift leads to a contradiction, so no shifts are possible and the assumed alignment of P and T in phase h is not possible, proving the lemma.

Since $h < i$, the right end of P will not be shifted in phase h past the right end of t_i; consequently, after the phase h shift a character of $\bar{P}$ is opposite character $T(k')$ (the character of T that will mismatch in phase i). Consider where the right end of P is after the phase h shift. There are two cases to consider: 1. Either the right end of P is opposite the right end of another full copy of β (in t_i) or 2. The right end of P is in the interior of a full copy of β.

Case 1 If the phase h shift aligns the right end of P with the right end of a full copy of β, then the character opposite $T(k')$ would be P(k $- r|\beta|$) for some r. But since $\bar{P}$ is

[1] **Later we will analyze the Boyer–Moore algorithm when P is in T. For that purpose we note here that when phase h is assumed to end by finding an occurrence of P, then the proof of Lemma 3.2.6 is complete at this point, having established a contradiction. That is, on the assumption that the right end of P is aligned with the right end of a β in phase h, we proved that phase h ends with a mismatch, which would contradict the assumption that h ends by finding an occurrence of P in T. So even if phase h ends by finding an occurrence of P, the right end of P could not be aligned with the right end of a β block in phase h.**

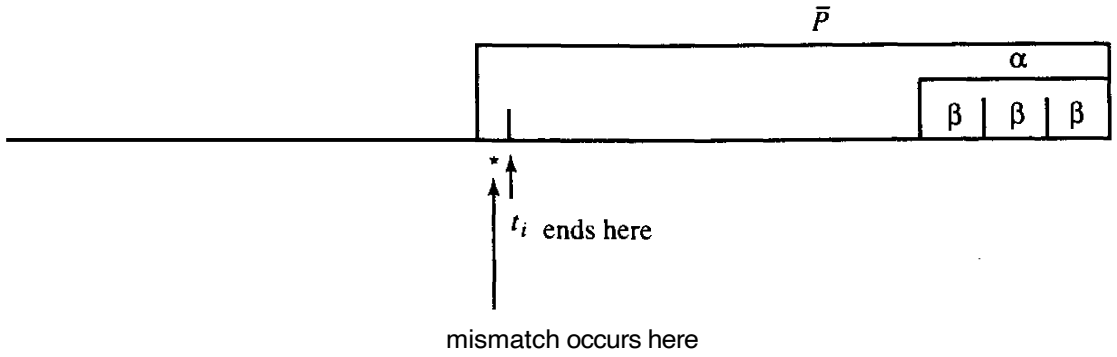

Figure 3.3: String α has length s_i; string $\bar{P}$ has length $|t_i| + 1$

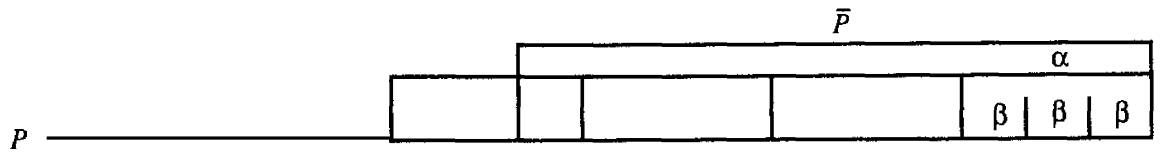

Figure 3.4: Starting from the right, substrings of length $|\alpha| = s_i$ are marked off in $\bar{P}$.

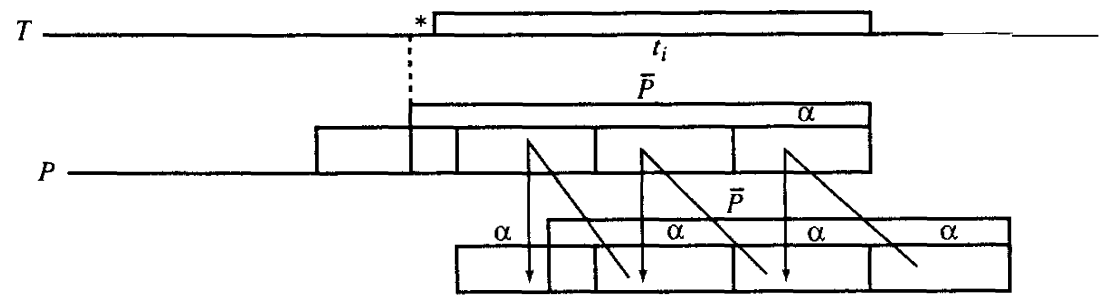

Figure 3.5: The arrows show the string equalities described in the proof.

of the shifted $\bar{P}$ below t_i must match the portion of the urrshifted $\bar{P}$ below ti, so the second marked-off substring from the right end of the shifted $\bar{P}$ must be the same as the first substring of the unshifted $\bar{P}$. Hence they must both be copies of string a. But the second substring is the same in both copies of $\bar{P}$, so continuing this reasoning we see that all the s_i-length marked substrings are copies of a and the left-most substring is a suffix of a (if it is not a complete copy of a) .Hence $\bar{P}$ is semiperiodic with period a. The right-most $|t_i|$ characters of $\bar{P}$ match t_i, and so t_i is also semiperiodic with period a. Then since $a = \beta^l$, $\bar{P}$ and t_i must also be semiperiodic with period β. □

Recall that we want to bound g_i, the number of characters compared in the i th phase that have been previously compared in earlier phases. All but one of the characters compared in phase i are contained in t_i, and a character in t_i could have previously been examined only during a phase where P overlaps t_i. So to bound g_i, we closely examine in what ways P could have overlapped t_i during earlier phases.

Lemma 3.2.6. If $|t_i| + 1 > 3s_i$, *then in any phase* $h < i$, *the right end of P could not have been aligned opposite the right end of any full copy of* β *in substring t, of* T.

PROOF By Lemma 3.2.5, t_i is semiperiodic with period β. Figure **3.6** shows string, t_i as a concatenation of copies of string β. In phase h, the right end of P cannot be aligned with the right end of t_i since that is the alignment of P and T in phase $i > h$, and P must have moved right between phases h and i. So, suppose, for contradiction, that in phase h the right end of P is aligned with the right end of some other full copy of β in t_i. For

so that the two characters of P aligned with $T(k'')$ before and after the shift are unequal. We claim these conditions hold when the right end of P is aligned with the right end of β'. Consider that alignment. Since $\bar{P}$ is semiperiodic with period β, that alignment of P and T would match at least until the left end of t_i and so would match at position k" of T. Therefore, the two characters of P aligned with $T(k'')$ before and after the shift cannot be equal. Thus if the end of P were aligned with the end of β' then all the characters of T that matched in phase h would again match, and the characters of P aligned with $T(k'')$ before and after the shift would be different. Hence the good suffix rule would not shift the right end of P past the right of the end of β'.

Therefore, if the right end of P is aligned in the interior of β' in phase h, it must also be in the interior of β' in phase h + 1. But h was arbitrary, so the phase-h + 1 shift would also not move the right end of P past β'. So if the right end of P is in the interior of β' in phase h, it remains there forever. This is impossible since in phase $i >$ h the right end of P is aligned with the right end of t_i, which is to the right of β'. Hence the right end of P is not in the interior of β', and the Lemma is proved. □

Note again that Lemma 3.2.8 holds even if phase h is assumed to end by finding an occurrence of P in T. That is, the proof only needs the assumption that phase i ends with a mismatch, not that phase h does. In fact, when phase h finds an occurrence of P in T, then the proof of the lemma only needs the reasoning contained in the first two paragraphs of the above proof.

Theorem 3.2.1. Assuming **P** does not occur in T, $s_i \geq g_i/3$ in every phase i.

PROOF This is trivially true if $s_i \geq (|t_i| + 1)/3$, so assume $|t_i| + 1 > 3s_i$. By Lemma 3.2.8, in any phase h $< i$, the right end of P is opposite either one of the left-most $|\beta| - 1$ characters of t_i or one of the right-most $|\beta|$ characters of t_i (excluding the extreme right character). By Lemma 3.2.7, at most $|\beta|$ comparisons are made in phase h $< i$. Hence the only characters compared in phase i that could possibly have been compared before phase i are the left-most $|\beta| - 1$ characters of t_i, the right-most $2|\beta|$ characters of t,, or the character just to the left of t_i. So $g_i \leq 3|\beta| = 3s_i$ when $|t_i| + 1 > 3s_i$. In both cases then, $s_i \geq g_i/3$. □

Theorem 3.2.2. [108] Assuming that P does not occur in T, the worst-case number *of* comparisons made by the Boyer–Moore algorithm is at most 4m.

PROOF As noted before, $\sum_{i=1}^{q} g_i' \leq$ m and $\sum_{i=1}^{q} s_i \leq$ m, so the total number of comparisons done by the algorithm is $\sum_{i=1}^{q}(g; + g_i') \leq (\sum_i 3s_i) + m \leq 4m$. □

3.2.2. The case when the pattern does occur in the text

Consider P consisting of n copies of a single character and T consisting of m copies of the same character. Then P occurs in T starting at every position in T except the last n − 1 positions, and the number of comparisons done by the Boyer–Moore algorithm is $\Theta(mn)$. The $O(m)$ time bound proved in the previous section breaks down because it was derived by showing that $g_i \leq 3s_i$, and that required the assumption that phase i ends with a mismatch. So when P does occur in T (and phases do not necessarily end with mismatches), we must modify the Boyer–Moore algorithm in order to recover the linear running time. Galil [168] gave the first such modification. Below we present a version of his idea.

The approach comes from the following observation: Suppose in phase i that the right end of P is positioned with character k of T, and that P is compared with T down

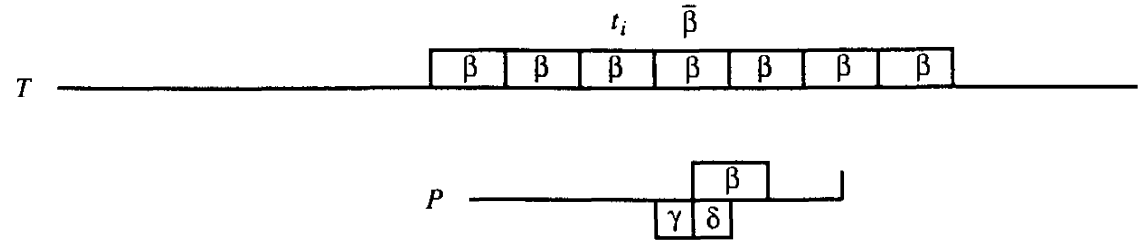

Figure 3.8: Case when the right end of P is aligned with a character in the interior of a β. Then t_i would have a smaller period than β, contradicting the definition of β.

serniperiodic with period β, $P(k)$ must be equal to P(k $- r|\beta|$), contradicting the good suffix rule.

Case 2 Suppose the phase h shift aligns P so that its right end aligns with some character in the interior of a full copy of β. That means that, in this alignment, the right end of some β string in P is opposite a character in the interior of $\bar{\beta}$. Moreover, by the good suffix rule, the characters in the shifted P below $\bar{\beta}$ agree with $\bar{\beta}$ (see Figure 3.8). Let $\gamma\delta$ be the string in the shifted P positioned opposite $\bar{\beta}$ in t_i, where γ is the string through the end of β and δ is the remainder. Since $\bar{\beta} = \beta$, γ is a suffix of β, δ is a prefix of β, and $|\gamma| + |\delta| = |\bar{\beta}| = 1/31$; thus $\gamma\delta = \delta\gamma$. By Lemma 3.2.1, however, $\beta = \rho^t$ for $t > 1$, which contradicts the assumption that β is the smallest string such that $\alpha = \beta^l$ for some l.

Starting with the assumption that in phase h the right end of P is aligned with the right end of a full copy of β, we reached the conclusion that no shift in phase h is possible. Hence the assumption is wrong and the lemma is proved. □

Lemma 3.2.7. *If $|t_i| + l > 3s_i$, then in phase $h < i$, P can match t_i in T for at most $|\beta| - 1$ characters.*

PROOF Since P is not aligned with the end of any β in phase h, if P matches t_i in T for β or more characters then the right-most β characters of P would match a string consisting of a suffix (γ) of β followed by a prefix (δ) of β. So we would again have $\beta = \gamma\delta = \delta\gamma$, and by Lemma 3.2.1, this again would lead to a contradiction to the selection of β. □

Note again that this lemma holds even if phase h is assumed to find an occurrence of P. That is, nowhere in the proof is it assumed that phase h ends with a mismatch, only that phase i does. This observation will be used later.

Lemma 3.2.8. *If $|t_i| + 1 > 3s_i$, then in phase $h < i$ if the right end of P is aligned with a character in t_i, it can only be aligned with one of the left-most $|\beta| - 1$ characters of t_i; or one of the right-most $|\beta|$ characters of t_i.*

PROOF Suppose in phase h that the right end of P is aligned with a character of t_i other than one of the left-most $|\beta| - 1$ characters or the right-most $|\beta|$ characters. For concreteness, say that the right end of P is aligned with a character in copy β' of string β. Since β' is not the left-most copy of β, the right end of P is at least $|\beta|$ characters to the right of the left end of t_i, and so by Lemma 3.2.7 a mismatch would occur in phase h before the left end of t_i is reached. Say that mismatch occurs at position k'' of T. After that mismatch, P is shifted right by some amount determined by the good suffix rule. By Lemma 3.2.6, the phase-h shift cannot move the right end of P to the right end of β', and we will show that the shift will also not move the end of P past the right end of β'.

Recall that the good suffix rule shifts P (when possible) by the smallest amount so that all the characters of T that matched in phase h again match with the shifted P and

all comparisons in phases that end with a mismatch have already been accounted for (in the accounting for phases not in Q) and are ignored here.

Let k' > k > i be a phase in which an occurrence of P is found overlapping the earlier run but is not part of that run. As an example of such an overlap, suppose P = *axaaxa* and T contains the substring *axaaxaaxaaxaxaaxaaxa*. Then a run begins at the start of the substring and ends with its twelfth character, and an overlapping occurrence of P (not part of the run) begins with that character. Even with the Galil rule, characters in the run will be examined again in phase k', and since phase k' does not end with a mismatch those comparisons must still be counted.

In phase k', if the left end of the new occurrence of P in T starts at a left end of a copy of β in the run, then contiguous copies of β continue past the right end of the run. But then no mismatch would have been possible in phase k since the pattern in phase k is aligned exactly $|\beta|$ places to the right of its position in phase k − 1 (where an occurrence of P was found). So in phase k', the left end of the new P in T must start with an interior character of some copy of β. But then if P overlaps with the run by more than $|\beta|$ characters, Lemma 3.2.1 implies that β is periodic, contradicting the selection of β. So P can overlap the run only by part of the run's left-most copy of β. Further, since phase k' ends by finding an occurrence of P, the pattern is shifted right by $s_{k'} = |\beta|$ positions. Thus any phase that finds an occurrence of P overlapping an earlier run next shifts P by a number of positions larger than the length of the overlap (and hence the number of comparisons). It follows then that over the entire algorithm the total number of such additional comparisons in overlapping regions is $O(m)$.

All comparisons are accounted for and hence $\sum_{i \in Q} d_i = O(m)$, finishing the proof of the lemma. □

3.2.3. Adding in the bad character rule

Recall that in computing a shift after a mismatch, the Boyer–Moore algorithm uses the largest shift given by either the (extended) bad character rule or the (strong) good suffix rule. It seems intuitive that if the time bound is $O(m)$ when only the good suffix rule is used, it should still be $O(m)$ when both rules are used. However, certain "interference" is plausible, and so the intuition requires a proof.

Theorem 3.2.4. When both shift *rules* are used *together*, the worst-case *running* time of the *modified* Boyer–Moore algorithm *remains* $O(m)$.

PROOF In the analysis using only the suffix rule we focused on the comparisons done in an arbitrary phase i. In phase i the right end of P was aligned with some character of T. However, we never made any assumptions about how P came to be positioned there. Rather, given an arbitrary placement of P in a phase ending with a mismatch, we deduced bounds on how many characters compared in that phase could have been compared in earlier phases. Hence all of the lemmas and analyses remain correct if P is arbitrarily picked up and moved some distance to the right at any time during the algorithm. The (extended) bad character rule only moves P to the right, so all lemmas and analyses showing the $O(m)$ bound remain correct even with its use. □

to character s of T. (We don't specify whether the phase ends by finding a mismatch or by finding an occurrence of P in T.) If the phase-i shift moves P so that its left end is to the right of character s of T, then in phase i + 1 a prefix of P definitely matches the characters of T up to $T(k)$. Thus, in phase i + 1, if the right-to-left comparisons get down to position k of T, the algorithm can conclude that an occurrence of P has been found even without explicitly comparing characters to the left of T(k + 1). It is easy to implement this modification to the algorithm, and we assume in the rest of this section that the Boyer–Moore algorithm includes this rule, which we call the Galil rule.

Theorem 3.2.3. Using the Galil rule, the Boyer–Moore algorithm never does more than $O(m)$ comparisons, no matter how many occurrences or P there are in T.

PROOF Partition the phases into those that do find an occurrence of P and those that do not. Let Q be the set of phases of the first type and let d_i be the number of comparisons done in phase i if $i \in Q$. Then $\sum_{i \in Q} d_i + \sum_{i \notin Q}(|t_i| + 1)$ is a bound on the total number of comparisons done in the algorithm.

The quantity $\sum_{i \notin Q}(|t_i| + 1)$ is again $O(m)$. To see this, recall that the lemmas of the previous section, which proved that $g_i \leq 3s_i$, only needed the assumption that phase i ends with a mismatch and that h $< i$. In particular, the analysis of how P of phase h $<$ i is aligned with P of phase i did not need the assumption that phase h ends with a mismatch. Those proofs cover both the case that h ends with a mismatch and that h ends by finding an occurrence of P. Hence it again holds that $g_i \leq 3s_i$ if phase i ends with a mismatch, even though earlier phases might end with a match.

For phases in Q, we again ignore the case that $s_i \geq (n + 1)/3 \geq (d_i + 1)/3$, since the total number of comparisons done in such phases must be bounded by $\sum 3s_i \leq 3m$. So suppose phase i ends by finding an occurrence of P in T and then shifts by less than $n/3$. By a proof essentially the same as for Lemma 3.2.5 it follows that P is semi-periodic; let β denote the shortest period of P. Hence the shift in phase i moves P right by exactly $|\beta|$ positions, and using the Galil rule in the Buyer–Moore algorithm, no character of T compared in phase i + 1 will have ever been compared previously. Repeating this reasoning, if phase i + 1 ends by finding an occurrence of P then P will again shift by exactly $|\beta|$ places and no comparisons in phase i + 2 will examine a character of T compared in any earlier phase. This cycle of shifting P by exactly $|\beta|$ positions and then identifying another occurrence of P by examining only $|\beta|$ new characters of T may be repeated many times. Such a succession of overlapping occurrences of P then consists of a concatenation of copies of β (each copy of P starts exactly $|\beta|$ places to the right of the previous occurrence) and is called a run. Using the Galil rule, it follows immediately that in any single run the number of comparisons used to identify the occurrences of P contained in that run is exactly the length of the run. Therefore, over the entire algorithm the number of comparisons used to find those occurrences is $O(m)$. If no additional comparisons were possible with characters in a run, then the analysis would be complete. However, additional examinations are possible and we have to account for them.

A run ends in some phase $k > i$ when a mismatch is found (or when the algorithm terminates), It is possible that characters of T in the run could be examined again in phases after k. A phase that reexamines characters of the run either ends with a mismatch or ends by finding an occurrence of P that overlaps the earlier run but is not part of it. However,

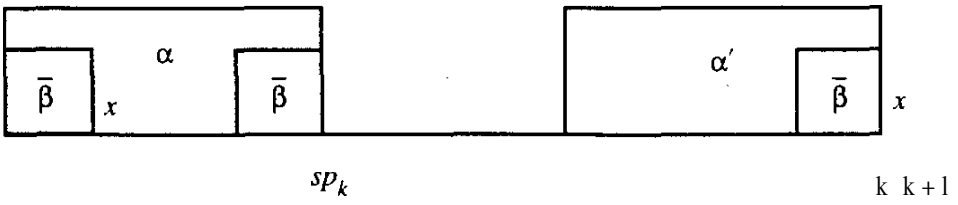

Figure 3.11: $\bar{\beta}$ must be a suffix of α.

$sp_k + 1 = |\alpha| + 1$, then $\bar{\beta}$ would be a prefix of P that is longer than a. But $\bar{\beta}$ is also a proper suffix of $P[1..k]$ (because βx is a proper suffix of $P[1..k+1]$). Those two facts would contradict the definition of sp_k (and the selection of a). Hence $sp_{k+1} \leq sp_k + 1$.

Now clearly, $sp_{k+1} = sp_k + 1$ if the character to the right of a is x, since a x would then be a prefix of P that also occurs as a proper suffix of $P[1..k+1]$. Conversely, if $sp_{k+1} = sp_k + 1$ then the character after a must be x. □

Lemma 3.3.1 identifies the largest "candidate" value for sp_{k+1} and suggests how to initially look for that value (and for string β). We should first check the character $P(sp_k+1)$, just to the right of a. If it equals $P(sp_k + 1)$ then we conclude that $\bar{\beta}$ equals a, β is ax, and sp_{k+1} equals $sp_k + 1$. But what do we do if the two characters are not equal?

3.3.3. The general case

When character P(k + 1) # $P(sp_k + 1)$, then $sp_{k+1} < sp_k + 1$ (by Lemma 3.3.1), so $sp_{k+1} \leq sp_k$. It follows that β must be a prefix of a, and $\bar{\beta}$ must be a *proper* prefix of α. Now substring $\beta = \bar{\beta}x$ ends at position k + 1 and is of length at most sp_k, whereas a' is a substring ending at position k and is of length sp_k. So $\bar{\beta}$ is a suffix of a', as shown in Figure 3.11. But since a' is a copy of a, $\bar{\beta}$ is also a suffix of a.

In summary, when P(k + I) # $P(sp_k + 1)$, $\bar{\beta}$ occurs as a suffix of a and also as a proper prefix of a followed by character x. So when P(k + 1) ≠ $P(sp_k + 1)$, $\bar{\beta}$ is the longest proper prefix of a that matches a suffix of α and that is followed by character x in position $|\bar{\beta}| + 1$ of P. See Figure 3.11.

However, since a = $P[1..sp_k]$, we can state this as

**) $\bar{\beta}$ is the longest proper prefix of $P[1..sp_k]$ that matches a suffix of $P[1..k]$ and that is followed by character x in position $|\bar{\beta}| + 1$ of P.

The general reduction

Statements * and ** differ only by the substitution of $P[1..sp_k]$ for $P[1..k]$ and are otherwise exactly the same. Thus, when $P(sp_k + 1)$ # P(k + I), the problem of finding $\bar{\beta}$ reduces to another instance of the original problem but on a smaller string ($P[1..sp_k]$ in place of $P[1..k]$). We should therefore proceed as before. That is, to search for $\bar{\beta}$ the algorithm should find the longest proper prefix of $P[1..sp_k]$ that matches a suffix of $P[1..sp_k]$ and then check whether the character to the right of that prefix is x. By the definition of sp_k, the required prefix ends at character sp_{sp_k}. So if character $P(sp_{sp_k}+1) = x$ then we have found $\bar{\beta}$, or else we recurse again, restricting our search to ever smaller prefixes of P. Eventually, either a valid prefix is found, or the beginning of P is reached. In the latter case, $sp_{k+1} = 1$ if $P(1) = P(k + 1)$; otherwise $sp_{k+1} = 0$.

The complete preprocessing algorithm

Putting all the pieces together gives the following algorithm for finding $\bar{\beta}$ and sp_{k+1}:

3.3. The original preprocessing for Knuth-Morris-Pratt

3.3.1. The method does not use fundamental preprocessing

In Section 1.3 we showed how to compute all the sp_i values from Z_i values obtained during fundamental preprocessing of P. The use of Z_i values was conceptually simple and allowed a uniform treatment of various preprocessing problems. However, the classical preprocessing method given in Knuth-Morris-Pratt [278] is not based on fundamental preprocessing. The approach taken there is very well known and is used or extended in several additional methods (such as the Aho–Corasick method that is discussed next). For those reasons, a serious student of string algorithms should also understand the classical algorithm for Knuth-Morris-Pratt preprocessing.

The preprocessing algorithm computes $sp_i(P)$ for each position i from i $= 2$ to $i = n$ (sp_1 is zero). To explain the method, we focus on how to compute sp_{k+1} assuming that sp_i is known for each i $\leq k$. The situation is shown in Figure 3.9, where string a is the prefix of P of length sp_k. That is, α is the longest string that occurs both as a proper prefix of P and as a substring of P ending at position k. For clarity, let α' refer to the copy of a that ends at position k.

Let x denote character k + 1 of P, and let $\beta = \bar{\beta}x$ denote the prefix of P of length sp_{k+1} (i.e., the prefix that the algorithm will next try to compute). Finding sp_{k+1} is equivalent to finding string $\bar{\beta}$. And clearly,

*) $\bar{\beta}$ is the longest proper prefix of P[1..k] that matches a suffix of P[1..k] and that is followed by character x in position $|\bar{\beta}| + 1$ of P. See Figure 3.10.

Our goal is to find sp_{k+1}, or equivalently, to find $\bar{\beta}$.

3.3.2. The easy case

Suppose the character just after a is x (i.e., $P(sp_k + 1) = x$). Then, string ax is a prefix of P and also a proper suffix of $P[1..k + 1]$, and thus $sp_{k+1} \geq |\alpha x| = sp_k + 1$. Can we then end our search for sp_{k+1} concluding that sp_{k+1} equals $sp_k + 1$, or is it possible for sp_{k+1} to be strictly greater than $sp_k + 1$? The next lemma settles this.

Lemma 3.3.1. *For any k, $sp_{k+1} \leq sp_k + 1$. Further; $sp_{k+1} = sp_k + 1$ if and only if the character after α is x. That is, $sp_{k+1} = sp_k + 1$ if and only if $P(sp_k + 1) = P(k + 1)$.*

PROOF Let $\beta = \bar{\beta}x$ denote the prefix of P of length sp_{k+1}. That is, $\beta = \bar{\beta}x$ is the longest proper suffix of $P[1..k + 1]$ that is a prefix of P. If sp_{k+1} is strictly greater than

Figure 3.9: The situation after finding sp_k.

Figure 3.10: sp_{k+1} is found by finding $\bar{\beta}$.

each time the for statement is reached; it is assigned a variable number of times inside the while loop, each time this loop is reached. Hence the number of times v is assigned is $n-1$ plus the number of times it is assigned inside the while loop. How many times that can be is the key question.

Each assignment of v inside the while loop must decrease the value of v, and each of the $n-1$ times v is assigned at the for statement, its value either increases by one or it remains unchanged (at zero). The value of v is initially zero, so the total amount that the value of v can increase (at the for statement) over the entire algorithm is at most $n-1$. But since the value of v starts at zero and is never negative, the total amount that the value of v can decrease over the entire algorithm must also be bounded by $n-1$, the total amount it can increase. Hence v can be assigned in the while loop at most $n-1$ times, and hence the total number of times that the value of v can be assigned is at most $2(n-1) = O(n)$, and the theorem is proved. □

3.3.4. How to compute the optimized shift values

The (stronger) sp'_i values can be easily computed from the sp_i values in $O(n)$ time using the algorithm below. For the purposes of the algorithm, character P(n + 1), which does not exist, is defined to be different from any character in P.

Algorithm SP'(P)

```
sp; = 0;
For i := 2 to n do
begin
    v := sp_i;
    If P(v + 1) # P(i + 1) then
        sp'_i := v
    else
        sp'_i := sp'_v;
end;
```

Theorem 3.3.2. Algorithm SP'(P) correctly computes all the sp'_i values in $O(n)$ time.

PROOF The proof is by induction on the value of i. Since $sp_1 = 0$ and $sp'_i \leq sp_i$ for all i, then sp', = 0, and the algorithm is correct for $i = 1$. Now suppose that the value of sp'_i set by the algorithm is correct for all i < k and consider i = k. If $P[sp_k + 1] \# P[k + 1]$ then clearly sp'_k is equal to sp_k, since the sp_k length prefix of $P[1..k]$ satisfies all the needed requirements. Hence in this case, the algorithm correctly sets sp'_k.

If $P(sp_k + 1) = P(k + 1)$, then $sp'_k < sp_k$ and, since $P[1..sp_k]$ is a suffix $P[1..k]$, sp'_k can be expressed as the length of the longest proper prefix of $P[1..sp_k]$ that also occurs as a suffix of $P[1..sp_k]$ with the condition that $P(k + 1) \# P(sp'_k + 1)$. But since $P(k + 1) = P(sp_k + 1)$, that condition can be rewritten as $P(sp_k + 1) \# P(sp'_k + 1)$. By the induction hypothesis, that value has already been correctly computed as sp'_{sp_k}. So when $P(sp_k + 1) = P(k + 1)$ the algorithm correctly sets sp'_k to sp'_{sp_k}.

Because the algorithm only does constant work per position, the total time for the algorithm is $O(n)$. □

It is interesting to compare the classical method for computing sp and sp' and the method based on fundamental preprocessing (i.e., on Z values). In the classical method the (weaker) sp values are computed first and then the more desirable sp' values are derived

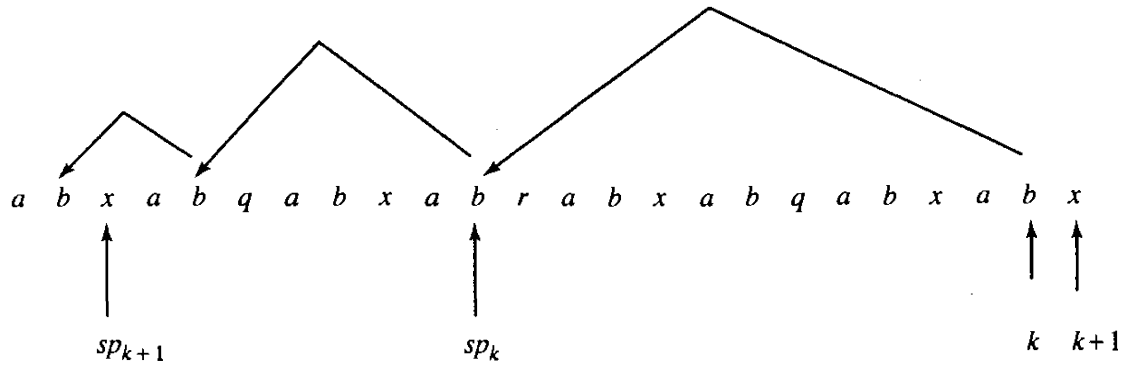

Figure 3.12: "Bouncing ball" cartoon of original Knuth-Morris-Pratt preprocessing. The arrows show the successive assignments to the variable v.

How to find sp_{k+1}

$x := P(k+1)$;
$v := sp_k$;
While $P(v+1) \# x$ and $v \# 0$ do
    $v := sp_v$;
end;
If $P(v+1) = x$ then
    $sp_{k+1} := v+1$
else
    $sp_{k+1} := 0$;

See the example in Figure 3.12.

The entire set of *sp* values are found as follows:

Algorithm SP(P)

$sp_1 = 0$
Fork:= 1 to n − 1 do
begin
    $x := P(k+1)$;
    $v := sp_k$;
    While $P(v+1) \# x$ and $v \neq 0$ do
        $v := sp_v$;
    end;
    If $P(v+1) =$ x then
        $sp_{k+1} := v+1$
    else
        $sp_{k+1} := 0$;
end;

Theorem 3.3.1. ***Algorithm SP finds all the $sp_i(P)$ values*** in $O(n)$ ***time, where n is the length of P.***

PROOF Note first that the algorithm consists of two nested loops, a ***for*** loop and a ***while*** loop. The ***for*** loop executes exactly $n - 1$ times, incrementing the value of k each time. The ***while*** loop executes a variable number of times each time it is entered.

The work of the algorithm is proportional to the number of times the value of v is assigned. We consider the places where the value of v is assigned and focus on how the value of v changes over the execution of the algorithm. The value of v is assigned once

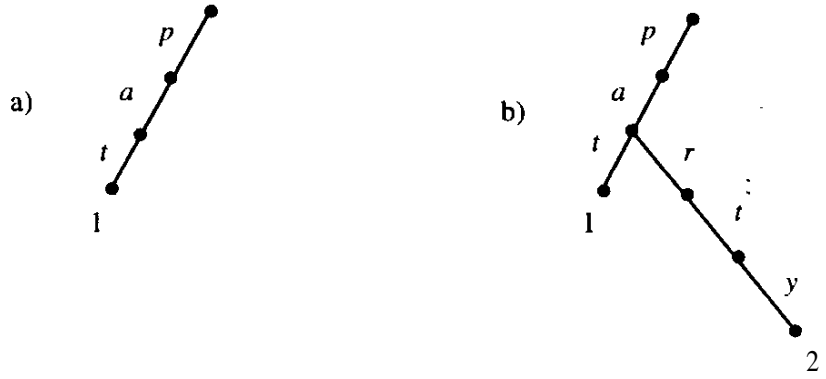

Figure 3.14: Pattern P_1 is the string ***pat.*** a. The insertion of pattern P_2 when P_2 is ***pa. b.*** The insertion when P_2 is ***party.***

Tree $\mathcal{K}_1$ just consists of a single path of $|P_1|$ edges out of root r. Each edge on this path is labeled with a character of P_1 and when read from the root, these characters spell out P_1. The number 1 is written at the node at the end of this path. To create $\mathcal{K}_2$ from $\mathcal{K}_1$, first find the longest path from root r that matches the characters of P_2 in order. That is, find the longest prefix of P_2 that matches the characters on some path from r. That path either ends by exhausting P_2 or it ends at some node v in the tree where no further match is possible. In the first case, P_2 already occurs in the tree, and so we write the number 2 at the node where the path ends. In the second case, we create a new path out of v, labeled by the remaining (unmatched) characters of P_2, and write number 2 at the end of that path. An example of these two possibilities is shown in Figure 3.14.

In either of the above two cases, $\mathcal{K}_2$ will have at most one branching node (a node with more than one child), and the characters on the two edges out of the branching node will be distinct. We will see that the latter property holds inductively for any tree $\mathcal{K}_i$. That is, at any branching node v in $\mathcal{K}_i$, all edges out of v have distinct labels.

In general, to create $\mathcal{K}_{i+1}$ from $\mathcal{K}_i$, start at the root of $\mathcal{K}_i$ and follow, as far as possible, the (unique) path in $\mathcal{K}_i$ that matches the characters in P_{i+1} in order. This path is unique because, at any branching node v of $\mathcal{K}_i$, the characters on the edges out of v are distinct. If pattern P_{i+1} is exhausted (fully matched), then number the node where the match ends with the number $i + 1$. If a node v is reached where no further match is possible but P_{i+1} is not fully matched, then create a new path out of v labeled with the remaining unmatched part of P_{i+1} and number the endpoint of that path with the number i + 1.

During the insertion of P_{i+1}, the work done at any node is bounded by a constant, since the alphabet is finite and no two edges out of a node are labeled with the same character. Hence for any i, it takes $O(|P_{i+1}|)$ time to insert pattern P_{i+1} into $\mathcal{K}_i$, and so the time to construct the entire keyword tree is $O(n)$.

3.4.1. Naive use of keyword trees for set matching

Because no two edges out of any node are labeled with the same character, we can use the keyword tree to search for all occurrences in T of patterns from $\mathcal{P}$. To begin, consider how to search for occurrences of patterns in $\mathcal{P}$ that begin at character 1 of T: Follow the unique path in $\mathcal{K}$ that matches a prefix of T as far as possible. If a node is encountered on this path that is numbered by i, then P_i occurs in T starting from position 1. More than one such numbered node can be encountered if some patterns in $\mathcal{P}$ are prefixes of other patterns in $\mathcal{P}$.

In general, to find all patterns that occur in T, start from each position 1 in T and follow the unique path from r in $\mathcal{K}$ that matches a substring of T starting at character l.

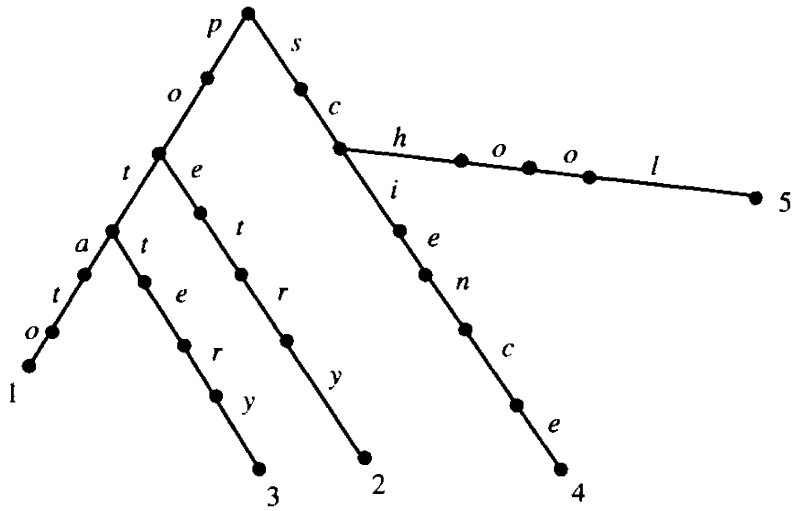

Figure 3.13: Keyword tree $\mathcal{K}$ with **five patterns.**

from them, whereas the order is just the opposite in the method based on fundamental preprocessing.

3.4. Exact matching with a set of patterns

An immediate and important generalization of the exact matching problem is to find all occurrences in text T of any pattern in a *set* of patterns $\mathcal{P} = \{P_1, P_2, \ldots, P_z\}$. This generalization is called the exact set matching problem. Let n now denote the total length of all the patterns in $\mathcal{P}$ and m be, as before, the length of T. Then, the exact set matching problem can be solved in time O(n + zm) by separately using any linear-time method for each of the z patterns.

Perhaps surprisingly, the exact set matching problem can be solved faster than, O(n + zm). It can be solved in O(n + m + k) time, where k is the number of occurrences in T of the patterns from P. The first method to achieve this bound is due to Aho and Corasick [9].[2] In this section, we develop the Aho–Corasick method; some of the proofs are left to the reader. An equally efficient, but more robust, method for the exact set matching problem is based on suffix trees and is discussed in Section 7.2.

Definition The keyword *tree* for set $\mathcal{P}$ is a rooted directed tree $\mathcal{K}$ satisfying three conditions: 1. each edge is labeled with exactly one character; 2. any two edges out of the same node have distinct labels; and 3. every pattern P_i in $\mathcal{P}$ maps to some node v of $\mathcal{K}$ such that the characters on the path from the coot of $\mathcal{K}$ to v exactly spell out P_i, and every leaf of $\mathcal{K}$ is mapped to by some pattern in P.

For example, Figure **3.13** shows the keyword tree for the set of patterns (potato, poetry, pottery, science, school).

Clearly, every node in the keyword tree corresponds to a prefix of one of the patterns in $\mathcal{P}$, and every prefix of a pattern maps to a distinct node in the tree.

Assuming a fixed-size alphabet, it is easy to construct the keyword tree for $\mathcal{P}$ in $O(n)$ time. Define $\mathcal{K}_i$ to be the (partial) keyword tree that encodes patterns $P_1, \ldots, P_i$ of $\mathcal{P}$.

[2] There is a more recent exposition of the Aho–Corasick method in [8], where the algorithm is used just as an "acceptor", deciding whether or not there is an occurrence in T of at least one pattern from $\mathcal{P}$. Because we will want to explicitly find all occurrences, that version of the algorithm is too limited to use here.

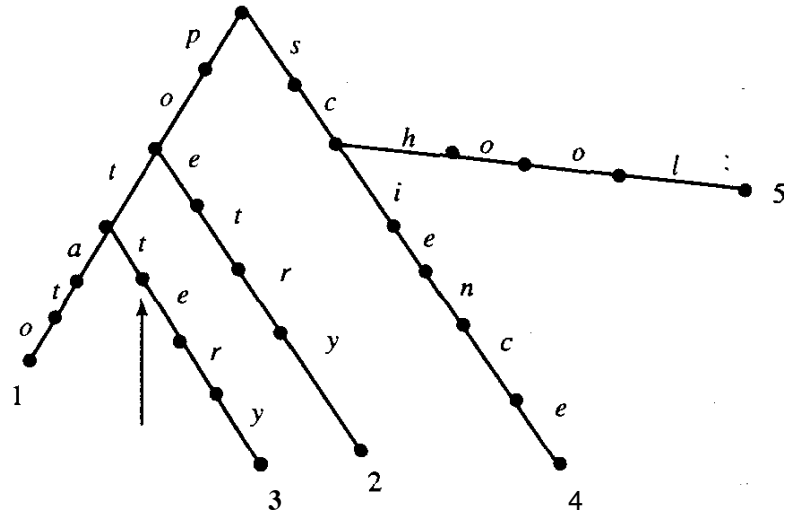

Figure 3.15: Keyword tree to illustrate the label of a node.

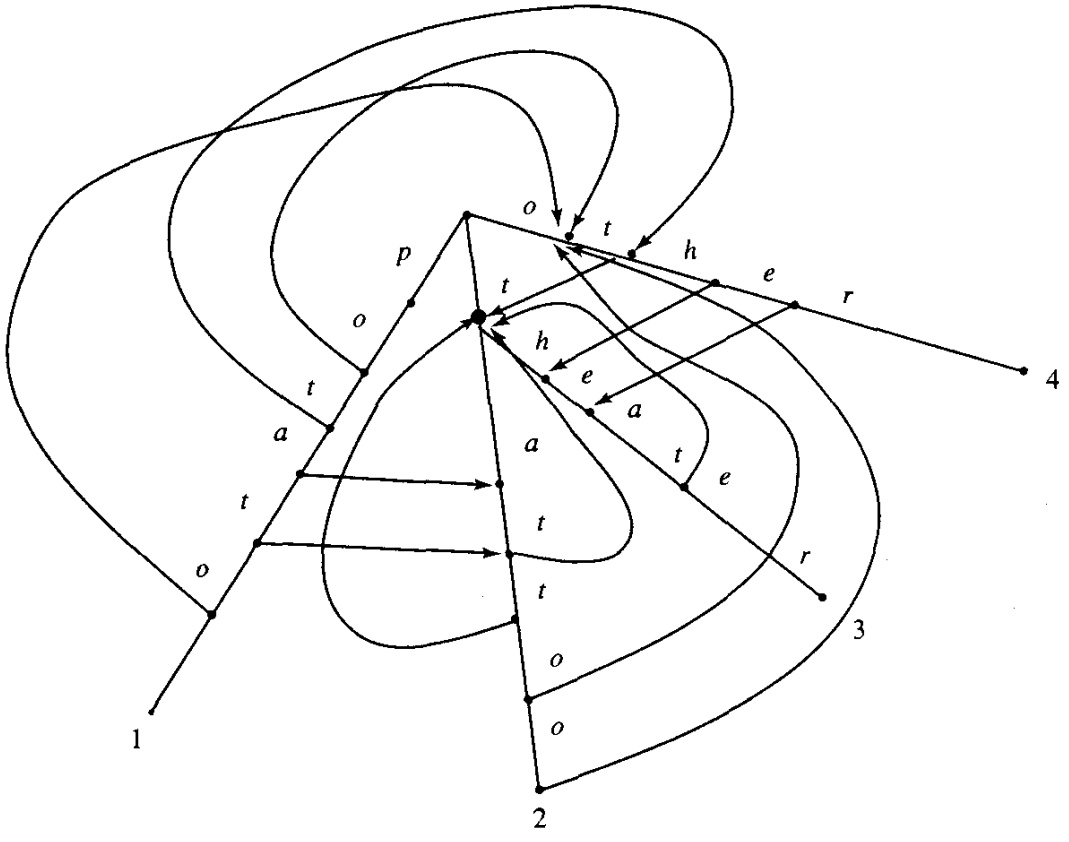

Figure 3.16: Keyword tree showing the failure links.

For example, consider the set of patterns $\mathcal{P} = \{potato,$ tattoo, *theater,* other$\}$ and its keyword tree shown in Figure 3.16. Let v be the node labeled with the string potar. Since *tat* is prefix of tattoo, and it is the longest proper suffix of *potat* that is a prefix of any pattern in $\mathrm{P}, lp(v) = 3$.

Lemma 3.4.1. *Let* a be the $lp(v)$*-length suffix of string* $\mathcal{L}(v)$. Then there is a unique node *in* the keyword tree that is labeled by string **a.**

PROOF $\mathcal{K}$ encodes all the patterns in $\mathcal{P}$ and, by definition, the $lp(v)$-length suffix of $\mathcal{L}(v)$ is a prefix of some pattern in $\mathcal{P}$. So there must be a path from the root in $\mathcal{K}$ that spells out

Numbered nodes along that path indicate patterns in $\mathcal{P}$ that start at position l. For a fixed l, the traversal of a path of $\mathcal{K}$ takes time proportional to the minimum of m and n, so by successively incrementing l from 1 to m and traversing $\mathcal{K}$ for each 1, the exact set matching problem can be solved in $O(nm)$ time. We will reduce this to O(n + m + k) time below, where k is the number of occurrences.

The dictionary problem

Without any further embellishments, this simple keyword tree algorithm efficiently solves a special case of set matching, called the dictionary problem. In the dictionary problem, a set of strings (forming a dictionary) is initially known and preprocessed. Then a sequence of individual strings will be presented; for each one, the task is to find if the presented string is contained in the dictionary. The utility of a keyword tree is clear in this context. The strings in the dictionary are encoded into a keyword tree $\mathcal{K}$, and when an individual string is presented, a walk from the root of $\mathcal{K}$ determines if the string is in the dictionary. Th this special case of exact set matching, the problem is to determine if the text T (an individual presented string) completely matches some string in $\mathcal{P}$.

We now return to the general set matching problem of determining which strings in $\mathcal{P}$ are contained in text T.

3.4.2. The speedup: generalizing Knuth-Morris-Pratt

The above naive approach to the exact set matching problem is analogous to the naive search we discussed before introducing the Knuth-Morris-Pratt method. Successively incrementing 1 by one and starting each search from root r is analogous to the naive exact match method for a single pattern, where after every mismatch the pattern is shifted by only one position, and the comparisons are always begun at the left end of the pattern. The Knuth-Morris-Pratt algorithm improves on that naive algorithm by shifting the pattern by more than one position when possible and by never comparing characters to the left of the current character in T. The Aho–Corasick algorithm makes the same kind of improvements, incrementing 1 by more than one and skipping over initial parts of paths in $\mathcal{K}$, when possible. The key is to generalize the function sp_i (defined on page 27 for a single pattern) to operate on a set of patterns. This generalization is fairly direct, with only one subtlety that occurs if a pattern in $\mathcal{P}$ is a proper substring of another pattern in $\mathcal{P}$. So, it is very helpful to (temporarily) make the following assumption:

Assumption No pattern in $\mathcal{P}$ is a proper substring of any other pattem in $\mathcal{P}$

3.4.3. Failure functions for the keyword tree

Definition Each node v in $\mathcal{K}$ is *labeled* with the string obtained by concatenating in order the characters on the path from the root of $\mathcal{K}$ to node v. $\mathcal{L}(v)$ is used to denote the label on v. That is, the concatenation of characters on the path from the root to v spells out the string $\mathcal{L}(v)$.

For example, in Figure 3.15 the node pointed to by the arrow is labeled with the string port.

Definition For any node v of $\mathcal{K}$, define $lp(v)$ to be the length of the longest proper suffix of string $\mathcal{L}(v)$ that is a prefix of some pattem in $\mathcal{P}$.

to the node n_v labeled *tat*, and $lp(v) = 3$. So 1 is incremented to $5 = 8 - 3$, and the next comparison is between character T(8) and character *t* on the edge below ***tat***.

With this algorithm, when no further matches are possible, l may increase by more than one, avoiding the reexamination of characters of T to the left of c, and yet we may be sure that every occurrence of a pattern in P that begins at character c $-$ $lp(v)$ of T will be correctly detected. Of course (just as in Knuth-Morris-Pratt), we have to argue that there are no occurrences of patterns of P starting strictly between the old l and $c - lp(v)$ in T, and thus l can be incremented to $c - lp(v)$ without missing any occurrences. With the given assumption that no pattern in $\mathcal{P}$ is a proper substring of another one, that argument is almost identical to the proof of Theorem 2.3.2 in the analysis of Knuth-Morris-Pratt, and it is left as an exercise.

When $lp(v) = 0$, then l is increased to c and the comparisons begin at the root of K. The only case remaining is when the mismatch occurs at the root. In this case, c must be incremented by 1 and comparisons again begin at the root.

Therefore, the use of function $v \mapsto n_v$ certainly accelerates the naive search for patterns of $\mathcal{P}$. But does it improve the worst-case running time? By the same sort of argument used to analyze the search time (not the preprocessing time) of Knuth-Morris-Pratt (Theorem 2.3.3), it is easily established that the search time for Aho–Corasick is $O(m)$. We leave this as an exercise. However, we have yet to show how to precompute the function $v \mapsto n_v$ in linear time.

3.4.5. Linear preprocessing for the failure function

Recall that for any node v of $\mathcal{K}$, n_v is the unique node in $\mathcal{K}$ labeled with the suffix of $\mathcal{L}(v)$ of length $lp(v)$. The following algorithm finds node n_v for each node v in $\mathcal{K}$, using $O(n)$ total time. Clearly, if v is the root r or v is one character away from r, then $n_v = r$. Suppose, for some k, n_v has been computed for every node that is exactly k or fewer characters (edges) from r. The task now is to compute n_v for a node v that is k + 1 characters from r. Let v' be the parent of v in $\mathcal{K}$ and let **x** be the character on the v' to v edge, as shown in Figure 3.17.

We are looking for the node n_v and the (unknown) string $\mathcal{L}(n_v)$ labeling the path to it from the root; we know node $n_{v'}$ because v' is k characters from r. Just as in the explanation

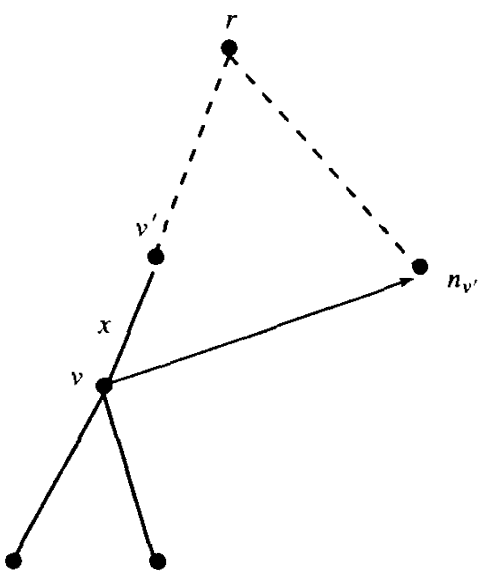

Figure 3.17: Keyword tree used to compute the failure function for node v.

string a. By the construction of $\mathcal{T}$ no two paths spell out the same string, so this path is unique and the lemma is proved. □

Definition For a node v of $\mathcal{K}$ let n_v be the unique node in $\mathcal{K}$ labeled with the suffix of $\mathcal{L}(v)$ of length $lp(v)$. When $lp(v) = 0$ then n_v is the root of $\mathcal{K}$.

Definition We call the ordered pair (u, n,) afailure *link.*

Figure 3.16 shows the keyword tree for $\mathcal{P}$ = {*potato,* tatoo, *theater,* other). Failure links are shown as pointers from every node v to node n_v where $lp(v) > 0$. The other failure links point to the root and are not shown.

3.4.4. The failure links speed up the search

Suppose that we know the failure link $v \mapsto n_v$ for each node v in $\mathcal{K}$. (Later we will show how to efficiently find those links.) How do the failure links help speed up the search? The Aho–Corasick algorithm uses the function $v \mapsto$ n, in a way that directly generalizes the use of the function i ↦ sp_i in the Knuth-Morris-Pratt algorithm. As before, we use l to indicate the starting position in T of the patterns being searched for. We also use pointer c into T to indicate the "current character" of T to be compared with a character on $\mathcal{K}$. The following algorithm uses the failure links to search for occurrences in T of patterns from $\mathcal{P}$:

Algorithm AC search

```
l := 1;
c := 1;
w := root of K;
repeat
    While there is an edge (w, w') labeled character T(c)
        begin
        if w' is numbered by pattern i then
            report that P_i occurs in T starting at position l;
        w := w' and c := c + 1;
        end;
    w := n_w and l := c - lp(w);
until c > m;
```

To understand the use of the function $v \mapsto n_v$, suppose we have traversed the tree to node v but cannot continue (i.e., character $T(c)$ does not occur on any edge out of v). We know that string $\mathcal{L}(v)$ occurs in T starting at position 1 and ending at position c − 1. By the definition of the function v ↦ n_v, it is guaranteed that string $\mathcal{L}(n_v)$ matches string $T[c - lp(v)..c - 1]$. That is, the algorithm could traverse $\mathcal{K}$ from the root to node n_v and be sure to match all the characters on this path with the characters in T starting from position $c - lp(v)$. So when $lp(v) \geq 0$,1 can be increased to $c - lp(v)$, c can be left unchanged, and there is no need to actually make the comparisons on the path from the root to node n_v. Instead, the comparisons should begin at node $a_{,,}$ comparing character c of T against the characters on the edges out of n_v.

For example, consider the text T = *xxpotattooxx* and the keyword tree shown in Figure 3.16. When l = **3,** the text matches the string *potat* but mismatches at the next character. At this point c = 8, and the failure link from the node v labeled *potat* points

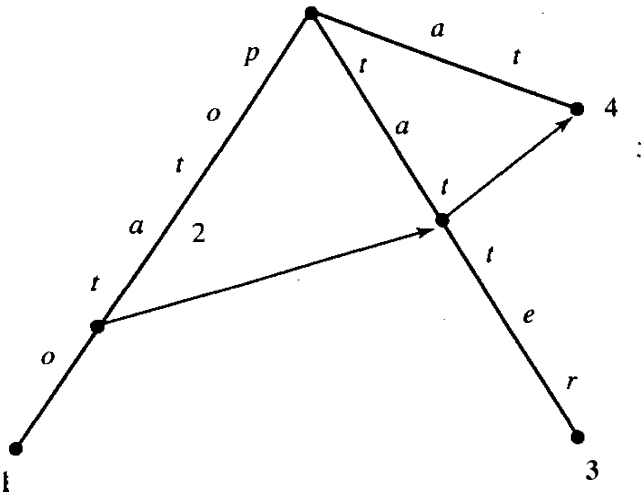

Figure 3.18: Keyword tree showing a directed path from *potat* to at through tat,

Repeating this analysis for every pattern in $\mathcal{P}$ yields the result that all the failure links are established in time proportional to the sum of the pattern lengths in $\mathcal{P}$ (i.e., in $O(n)$ total time). □

3.4.6. The full Aho–Corasick algorithm: relaxing the substring assumption

Until now we have assumed that no pattern in $\mathcal{P}$ is a substring of another pattern in $\mathcal{P}$. We now relax that assumption, If one pattern is a substring of another, and yet ***Algorithm AC search*** (page 56) uses the same keyword tree as before, then the algorithm may make l too large. Consider the case when $\mathcal{P} = \{acatt, ca\}$ and $T = acatg$. As given, the algorithm matches T along a path in $\mathcal{K}$ until character g is the current character, That path ends at the node v with $\mathcal{L}(v) = acat$. Now no edges out of v are labeled g, and since no proper suffix of *acat* is a prefix of *acatt* or ac, n_v is the root of $\mathcal{K}$. So when the algorithm gets stuck at node v it returns to the root with g as the current character, and it sets l to 5. Then after one additional comparison the current character pointer will be set to $m + 1$ and the algorithm will terminate without finding the occurrence of *ca* in T. This happens because the algorithm shifts (increases l) so as to match the longest *suffix* of $\mathcal{L}(v)$ with a prefix of some pattern in P. Embedded occurrences of patterns in $\mathcal{L}(v)$ that are not suffixes of $\mathcal{L}(v)$ have no influence on how much l increases.

It is easy to repair this problem with the following observations whose proofs we leave to the reader.

Lemma 3.4.2. ***Suppose in a keyword tree $\mathcal{K}$ there is a directed path of failure links (possibly empty) from a node v to a node that is numbered with pattern i. Then pattern P_i must occur in T ending at position c (the current character) whenever node v is reached during the search phase of the Aho–Corasick algorithm.***

For example, Figure 3.18 shows the keyword tree for $\mathcal{P} = (potato, pot, tatter, at)$ along with some of the failure links. Those links form a directed path from the node v labeled *potat* to the numbered node labeled *at*. If the traversal of $\mathcal{K}$ reaches v then T certainly contains the patterns *tat* and *at* end at the current c.

Conversely,

Lemma 3.4.3. ***Suppose a node v has been reached during the algorithm. Then pattern***

of the classic preprocessing for Knuth-Morris-Pratt, $\mathcal{L}(n_v)$ must be a suffix of $\mathcal{L}(n_{v'})$ (not necessarily proper) followed by character x. So the first thing to check is whether there is an edge ($n_{v'}$, w') out of node $n_{v'}$ labeled with character x. If that edge does exist, then n_v is node w' and we are done. If there is no such edge out of $n_{v'}$ labeled with character x, then $\mathcal{L}(n_v)$ is a proper suffix of $\mathcal{L}(n_{v'})$ followed by x. So we examine $n_{n_{v'}}$ next to see if there is an edge out of it labeled with character x. (Node $n_{n_{v'}}$ is known because $n_{v'}$ is k or fewer edges from the root.) Continuing in this way, with exactly the same justification as in the classic preprocessing for Knuth-Morris-Pratt, we arrive at the following algorithm for computing n_v for a node v:

Algorithm n_v

v' is the parent of v in $\mathcal{K}$;
x is the character on the edge (v', v);
w := $n_{v'}$;
While there is no edge out of w labeled x and w # r
    do w := n,;
end (while);
If there is an edge (w, w') out of w labeled x then
    $n_v := w'$;
else
    n_v := r;

Note the importance of the assumption that n_u is already known for every node u that is k or fewer characters from r.

To find n_v for every node v, repeatedly apply the above algorithm to the nodes in $\mathcal{K}$ in a breadth-first manner starting at the root.

Theorem 3.4.1. *Let* n be the total length of all the patterns in $\mathcal{P}$. The total time used by Algorithm n, when applied to all nodes in $\mathcal{K}$ is O(n).

PROOF The argument is a direct generalization of the argument used to analyze time in the classic preprocessing for Knuth-Morris-Pratt. Consider a single pattern P in $\mathcal{P}$ of length t and its path in $\mathcal{K}$ for pattern P. We will analyze the time used in the algorithm to find the failure links for the nodes on this path, as if the path shares no nodes with paths for any other pattern in $\mathcal{P}$. That analysis will overcount the actual amount of work done by the algorithm, but it will still establish a linear time bound.

The key is to see how $lp(v)$ varies as the algorithm is executed on each successive node v down the path for P. When v is one edge from the root, then $lp(v)$ is zero. Now let v be an arbitrary node on the path for P and let v' be the parent of v. Clearly, $lp(v) \leq lp(v') + 1$, so over all executions of Algorithm n, for nodes on the path for P, $lp()$ is increased by a total of at most t. Now consider how $lp()$ can decrease. During the computation of n_v for any node v, w starts at $n_{v'}$ and so has initial node depth equal to $lp(v')$. However, during the computation of n_v, the node depth of w decreases every time an assignment to w is made (inside the while loop). When n_v is finally set, $lp(v)$ equals the current depth of w, so if w is assigned k times, then $lp(v) \leq lp(v') - k$ and $lp()$ decreases by at least k. Now $lp()$ is never negative, and during all the computations along path P, $lp()$ can be increased by a total of at most t. It follows that over all the computations done for nodes on the path for P, the number of assignments made inside the while loop is at most t. The total time used is proportional to the number of assignments inside the loop, and hence all failure links on the path for P are set in $O(t)$ time.

of an output link leads to the discovery of a pattern occurrence, so the total time for the algorithm is $O(n+m+k)$, where k is the total number of occurrences. In summary we have,

Theorem 3.4.2. *If $\mathcal{P}$ is a set of patterns with total length n and T is a text of total length m, then one can find all occurrences in T of patterns from P in $O(n)$ preprocessing time plus $O(m+k)$ search time, where k is the number of occurrences. This is true even without assuming that the patterns in P are substring free.*

In a later chapter (Section 6.5) we will discuss further implementation issues that affect the practical performance of both the Aho–Corasick method, and suffix tree methods.

3.5. Three applications of exact set matching

3.5.1. Matching against a DNA or protein library of known patterns

There are a number of applications in molecular biology where a relatively stable library of interesting or distinguishing DNA or protein substrings have been constructed. The ***Sequence-tagged sites*** (STSs) and ***Expressed sequence tugs*** (ESTs) provide our first important illustration.

Sequence-tagged-sites

The concept of a Sequence-tagged-site (STS) is one of the most useful by-products that has come out of the Human Genome Project [111, 234, 399]. Without going into full biological detail, an STS is intuitively a DNA string of length 200–300 nucleotides whose right and left ends, of length 20–30 nucleotides each, occur only once in the entire genome [111, 317]. Thus each STS occurs uniquely in the DNA of interest. Although this definition is not quite correct, it is adequate for our purposes. An early goal of the Human Genome Project was to select and map (locate on the genome) a set of STSs such that any substring in the genome of length 100,000 or more contains at least one of those STSs. A more refined goal is to make a map containing ESTs (expressed sequence tags), which are STSs that come from genes rather than parts of intergene DNA. ESTs are obtained from mRNA and cDNA (see Section 11.8.3 for more detail on cDNA) and typically reflect the protein coding parts of a gene sequence.

With an STS map, one can locate on the map any sufficiently long string of anonymous but sequenced DNA – the problem is just one of finding which STSs are contained in the anonymous DNA. Thus with STSs, map location of anonymous sequenced DNA becomes a string problem, an exact set matching problem. The STSs or the ESTs provide a computer-based set of indices to which new DNA sequences can be referenced. Presently, hundreds of thousands of STSs and tens of thousands of ESTs have been found and placed in computer databases [234]. Note that the total length of all the STSs and ESTs is very large compared to the typical size of an anonymous piece of DNA. Consequently, the keyword tree and the Aho–Corasick method (with a search time proportional to the length of the anonymous DNA) are of direct use in this problem for they allow very rapid identification of STSs or ESTs that occur in newly sequenced DNA.

Of course, there may be some errors in either the STS map or in the newly sequenced DNA causing trouble for this approach (see Section 16.5 for a discussion of STS maps). But in this application, the number of errors should be a small percentage of the length of the STS, and that will allow more sophisticated exact (and inexact) matching methods to succeed. We will describe some of these in Sections 7.8.3, 9.4, and 12.2 of the book.

P_i occurs in T ending at position c only if v is numbered i or there is a directed path of failure links from v to the node numbered i.

So the full search algorithm is

Algorithm full AC search

```
l := 1;
c := 1;
w := root;
repeat
    While there is an edge (w, w') labeled T(c)
        begin
        if w' is numbered by pattern i or there is
        a directed path of failure links from w' to a node numbered with i
            then report that P_i occurs in T ending at position c;
        w := w' and c := c + 1;
        end;
    w := n_w and l := c - lp(w);
until c > n;
```

Implementation

Lemmas 3.4.2 and 3.4.3 specify at a high level how to find all occurrences of the patterns in the text. but specific implementation details are still needed. The goal is to be able to build the keyword tree, determine function $v \mapsto n_v$, and be able to execute the full AC search algorithm all in $O(m + k)$ time. To do this we add an additional pointer, called the ***output link,*** to each node of $\mathcal{K}$.

The output link (if there is one) at a node v points to that numbered node (a node associated with the end of a pattern in P) other than v that is reachable from v by the fewest failure links. The output links can be determined in $O(n)$ time during the running of the preprocessing algorithm n_v. When the n_v value is determined, the possible output link from node v is determined as follows: If n_v is a numbered node then the output link from v points to n,; if n, is not numbered but has an output link to a node w, then the output link from v points to w; otherwise v has no output link. In this way, an output link points only to a numbered node, and the path of output links from any node v passes through all the numbered nodes reachable from v via a path of failure links. For example, in Figure 3.18 the nodes for *tat* and *potat* will have their output links set to the node for *at,* The work of adding output links adds only constant time per node, so the overall time for algorithm n_v remains $O(n)$.

With the output links, all occurrences in T of patterns of $\mathcal{P}$ can be detected in $O(m + k)$ time. As before, whenever a numbered node is encountered during the full AC search, an occurrence is detected and reported. But additionally, whenever a node v is encountered that has an output link from it, the algorithm must traverse the path of output links from v, reporting an occurrence ending at position c of T for each link in the path. When that path traversal reaches a node with no output link, it returns along the path to node v and continues executing the full AC search algorithm. Since no character comparisons are done during any output link traversal, over both the construction and search phases of the algorithm the number of character comparisons is still bounded by $O(n+m)$. Further, even though the number of traversals of output links can exceed that linear bound, each traversal

text T. For each starting location j of P_i in T,
increment the count in cell $j - l_i + 1$ of C by one.

{For example, if the second copy of string ab is found in T
starting at position 18, then cell 12 of C is incremented by one.)

3. Scan vector C for any cell with value k. There is an occurrence of P
in T starting at position p if and only if $C(p) = k$.

Correctness and complexity of the method

Correctness Clearly, there is an occurrence of P in T starting at position p if and only if, for each i, subpattern $P_i \in \mathrm{P}$ occurs at position $j = \mathrm{p} + l_i - 1$ of T. The above method uses this idea in reverse. If pattern $P_i \in \mathrm{P}$ is found to occur starting at 'position j of T, and pattern P_i starts at position l_i in P, then this provides one "witness" that P occurs at T starting at position $p = j - l_i + 1$. Hence P occurs in T starting at p if and only if similar witnesses for position p are found for each of the k strings in P. The algorithm counts, at position p, the number of witnesses that observe an occurrence of P beginning at p. This correctly determines whether P occurs starting at p because each string in $\mathcal{P}$ can cause at most one increment to cell p of C.

Complexity The time used by the Aho–Corasick algorithm to build the keyword tree for $\mathcal{P}$ is $O(n)$. The time to search for occurrences in T of patterns from P is $O(m + z)$, where $|T| = m$ and z is the number of occurrences. We treat each pattern in P as being distinct even if there are multiple copies of it in $\mathcal{P}$. Then whenever an occurrence of a pattern from $\mathcal{P}$ is found in T, exactly one cell in C is incremented; furthermore, a cell can be incremented to at most k. Hence z must be bounded by km, and the algorithm runs in $O(km)$ time . Although the number of character comparisons used is just $O(m)$, km need not be $O(m)$ and hence the number of times C is incremented may grow faster than $O(m)$, leading to a nonlinear $O(km)$ time bound. But if k is assumed to be bounded (independent of |P|), then the method does run in linear time. In summary,

Theorem **3.5.1.** ***If the number of wild cards in pattern P is bounded by a constant, then the exact matching problem with wild cards in the Pattern can be solved in $O(n+m)$ time.***

Later, in Sections 9.3, we will return to the problem of wild cards when they occur in either the pattern, text, or both.

3.5.3. Two-dimensional exact matching

A second classic application of exact set matching occurs in a generalization of string matching to two-dimensional exact matching. Suppose we have a ***rectangular*** digitized picture T, where each point is given a number indicating its color and brightness. We are also given a smaller rectangular picture P, which also is digitized, and we want to find all occurrences (possibly overlapping) of the smaller picture in the larger one. We assume that the bottom edges of the two rectangles are parallel to each other. This is a two-dimensional generalization of the exact string matching problem.

Admittedly, this problem is somewhat contrived. Unlike the one-dimensional exact matching problem, which truly arises in numerous practical applications, compelling applications of two-dimensional exact matching are hard to find. Two-dimensional matching that is inexact, allowing some errors, is a more realistic problem, but its solution requires

A related application comes from the "BAC-PAC" proposal [442] for sequencing the human genome (see page 418). In that method, 600,000 strings (patterns) of length 500 would first be obtained and entered into the computer. Thousands of times thereafter, one would look for occurrences of any of these 600,000 patterns in text strings of length 150,000. Note that the total length of the patterns is 300 million characters, which is two-thousand times as large as the typical text to be searched.

3.5.2. Exact matching with wild cards

As an application of exact set matching, we return to the problem of exact matching with a single pattern, but complicate the problem a bit. We modify the exact matching problem by introducing a character ϕ, called a wild card, that matches any single character. Given a pattern P containing wild cards, we want to find all occurrences of P in a text T. For example, the pattern $ab\phi\phi c\phi$ occurs twice in the text xabvccbababcax. Note that in this version of the problem no wild cards appear in T and that each wild card matches only a single character rather than a substring of unspecified length.

The problem of matching with wild cards should need little motivating, as it is not difficult to think up realistic cases where the pattern contains wild cards. One very important case where simple wild cards occur is in DNA transcription factors. A transcription factor is a protein that binds to specific locations in DNA and regulates, either enhancing or suppressing, the transcription of the DNA into RNA. In this way, production of the protein that the DNA codes for is regulated. The study of transcription factors has exploded in the past decade; many transcription factors are now known and can be separated into families characterized by specific substrings containing wild cards. For example, the Zinc Finger is a common transcription factor that has the following signature:

$$\text{CYS}\phi\phi\text{CYS}\phi\phi\phi\phi\phi\phi\phi\phi\phi\phi\phi\phi\text{HIS}\phi\phi\text{HIS},$$

where **CYS** is the amino acid cysteine and HIS is the amino acid histidine. Another important transcription factor is the *Leucine* Zipper, which consists of four to seven leucines, each separated by six wild card amino acids.

If the number of permitted wild cards is unbounded, it is not known if the problem can be solved in linear time. However, if the number of wild cards is bounded by a fixed constant (independent of the size of P) then the following method, based on exact set pattern matching, runs in linear time:

Exact matching **with** wild cards

0. Let C be a vector of length $|T|$ initialized to all zeros.

1. Let $\mathcal{P} = \{P_1, P_2, \ldots, P_k\}$ be the (multi-)set of maximal substrings of P that do not contain any wild cards. Let $l_1, l_2, \ldots, l_k$ be the starting positions in P of each of these substrings.

(For example, if P $= ab\phi\phi c\phi ab\phi\phi$ then $\mathcal{P} =$ (ab, c, ab) and $l_1 = 1, l_2 = 5, l_3 = 7$,)

2. Using the **Aho–Corasick** algorithm (or the suffix tree approach to be discussed later), find for each string P_i in $\mathcal{P}$, all starting positions of P_i in

3.6. Regular expression pattern matching

A regular expression is a way to specify a set of related strings, sometimes referred to as a *pattern.*[3] Many important sets of substrings (patterns) found in biosequences, particularly in proteins, can be specified as regular expressions, and several databases have been constructed to hold such patterns. The PROSITE database, developed by Amos Bairoch [41,421, is the major regular expression database for significant patterns in proteins (see Section 15.8 for more on PROSITE).

In this section, we examine the problem of finding substrings of a text string that match one of the strings specified by a given regular expression. These matches are computed in the Unix utility grep, and several special programs have been developed to find matches to regular expressions in biological sequences [279, 416, 422].

It is helpful to start first with an example of a simple regular expression. A formal definition of a regular expression is given later. The following PROSITE expression specifies a set of substrings, some of which appear in a particular family of granin proteins:

[ED]-[EN]-L-[SAN]-*x*-*x*-[DE]-*x*-E-L.

Every string specified by this regular expression has ten positions, which are separated by a dash. Each capital letter specifies a single amino acid and a group of amino acids enclosed by brackets indicates that exactly one of those amino acids must be chosen. A small *x* indicates that any one of the twenty amino acids from the protein alphabet can be chosen for that position. This regular expression describes 192,000 amino acid strings, but only a few of these actually appear in any known proteins. For example, ENLSSEDEEL is specified by the regular expression and is found in human granin proteins.

3.6.1. Formal definitions

We now give a formal, recursive definition for a regular expression formed from an alphabet Σ. For simplicity, and contrary to the PROSITE example, assume that alphabet C does not contain any symbol from the following list: $*, +, (,), \epsilon$.

Definition A single character from C is a regular expression. The symbol ϵ is a regular expression. A regular expression followed by another regular expression is a regular expression. Two regular expressions separated by the symbol "+" form a regular expression. A regular expression enclosed in parentheses is a regular expression. A regular expression enclosed in parentheses and followed by the symbol "*" is a regular expression. The symbol * is called the Kleene closure.

These recursive rules are simple to follow, but may need some explanation. The symbol ϵ represents the empty string (i.e., the string of length zero). If R is a parenthesized regular expression, then R^* means that the expression R can be repeated any number of times (including zero times). The inclusion of parentheses as part of a regular expression (outside of C) is not standard, but is closer to the way that regular expressions are actually specified in many applications. Note that the example given above in PROSITE format does not conform to the present definition but can easily be converted to do so.

As an example, let Σ be the alphabet of lower case English characters. Then $R = (a + c + t)ykk(p + q)^* \; vdt(l + z + \epsilon)(pq)$ is a regular expression over Σ, and $S =$

[3] Note that in the context of regular expressions, the meaning of the word "pattern" is different from its previous and general meaning in this book.

more complex techniques of the type we will examine in Part III of the book. So for now, we view two-dimensional exact matching as an illustration of how exact set matching can be used in more complex settings and as an introduction to more realistic two-dimensional problems. The method presented follows the basic approach given in [44] and [66]. Since then, many additional methods have been presented since that improve on those papers in various ways. However, because the problem as stated is somewhat unrealistic, we will not discuss the newer, more complex, methods. For a sophisticated treatment of two-dimensional matching see [22] and [169].

Let m be the total number of points in T, let n be the number of points in P, and let n' be the number of rows in P. Just as in exact string matching, we want to find the smaller picture in the larger one in O(n + m) time, where $O(nm)$ is the time for the obvious approach. Assume for now that each of the rows of P are distinct; later we will relax this assumption.

The method is divided into two phases. In the first phase, search for all occurrences of each of the rows of P among the rows of T. To do this, add an end of row marker (some character not in the alphabet) to each row of T and concatenate these rows together to form a single text string T' of length $O(m)$. Then, treating each row of P as a separate pattern, use the Aho–Corasick algorithm to search for all occurrences in T' of any row of P. Since P is rectangular, all rows have the same width, and so no row is a proper substring of another and we can use the simpler version of Aho–Corasick discussed in Section 3.4.2. Hence the first phase identifies all occurrences of complete rows of P in complete rows of T and takes O(n + m) time.

Whenever an occurrence of row i of P is found starting at position (p, q) of T, write the number i in position (p, q) of another array M with the same dimensions as T. Because each row of P is assumed to be distinct and because P is rectangular, at most one number will be written in any cell of M.

In the second phase, scan each column of M, looking for an occurrence of the string $1, 2, \ldots, n'$ in consecutive cells in a single column. For example, if this string is found in column 6, starting at row 12 and ending at row n' + 12, then P occurs in T when its upper left corner is at position (6,12). Phase two can be implemented in $O(n' + m) = O(n + m)$ time by applying any linear-time exact matching algorithm to each column of M.

This gives an $O(n + m)$ time solution to the two-dimensional exact set matching problem. Note the similarity between this solution and the solution to the exact matching problem with wild cards discussed in the previous section. A distinction will be discussed in the exercises.

Now suppose that the rows of P are not all distinct. Then, first find all identical rows and give them a common label (this is easily done during the construction of the keyword tree for the row patterns). For example, if rows 3, 6, and 10 are the same then we might give them all the label of 3. We do a similar thing for any other rows that are identical. Then, in phase one, only look for occurrences of row 3, and not rows 6 and 10. This ensures that a cell of M will have at most one number written in it during phase 1. In phase 2, don't look for the string 1, 2, 3, . . . , n' in the columns of M, but rather for a string where 3 replaces 6 and 10, etc. It is easy to verify that this approach is correct and that it takes just $O(n + m)$ time. In summary,

Theorem 3.5.2. *If T* and P are rectangular pictures with m and n cells, respectively, then all exact occurrences of P *in* T can be *found* in $O(n + m)$ time, improving upon the naive method, which takes $O(nm)$ time.

3.7. Exercises

1. Evaluate empirically the speed of the Boyer–Moore method against the Apostolic+ Giancarlo method under different assumptions about the text and the pattern. These assumptions should include the size of the alphabet, the "randomness" of the text or pattern, the level of periodicity of the text or pattern, etc.
2. In the Apostolico–Giancarlo method, array M is of size m, which may be large. Show how to modify the method so that it runs in the same time, but in place of M uses an array of size n.
3. In the Apostolico–Giancarlo method, it may be better to compare the characters first and then examine M and N if the two characters match. Evaluate this idea both theoretically and empirically.
4. In the Apostolico–Giancarlo method, $M(j)$ is set to be a number less than or equal to the length of the (right-to-left) match of P and T starting at position j of T. Find examples where the algorithm sets the value to be strictly less than the length of the match. Now, since the algorithm learns the exact location of the mismatch in all cases, $M(j)$ could always be set to the full length of the match, and this would seem to be a good thing to do. Argue that this change would result in a correct simulation of Boyer–Moore. Then explain why this was not done in the algorithm.

   Hint: tt's the time bound.
5. Prove Lemma 3.2.2 showing the equivalence of the two definitions of semiperiodic strings.
6. For each of the n prefixes of P, we want to know whether the prefix $P[1..i]$ is a periodic string. That is, for each i we want to know the largest $k > 1$ (if there is one) such that $P[1..i]$ can be written as a^k for some string a. Of course, we also want to know the period. Show how to determine this for all n prefixes in time linear in the length of P.

   Hint: Z-algorithm.
7. Solve the same problem as above but modified to determine whether each prefix is semiperiodic and with what period. Again, the time should be linear.
8. By being more careful in the bookkeeping, establish the constant in the $O(m)$ bound from Cole's linear-time analysis of the Boyer–Moore algorithm.
9. Show where Cole's worst-case bound breaks down if only the weak Boyer–Moore shift rule is used. Can the argument be fixed, or is the linear time bound simply untrue when only the weak rule is used? Consider the example of $T = abababababababababab$ and $P = xaaaaaaaaa$ without also using the bad character rule.
10. Similar to what was done in Section 1.5, show that applying the classical Knuth-Morris-Pratt preprocessing method to the string $P\$T$ gives a linear-time method to find all occurrence of P in T. In fact, the search part of the Knuth-Morris-Pratt algorithm (alter the preprocessing of P is finished) can be viewed as a slightly optimized version of the Knuth-Morris-Pratt preprocessing algorithm applied to the T part of $P\$T$. Make this precise, and quantify the utility of the optimization.
11. Using the assumption that $\mathcal{P}$ is *substring* free (i.e., that no pattern $P_i \in \mathcal{P}$ is a substring of another pattern $P_j \in P$), complete the correctness proof of the Aho–Corasick algorithm. That is, prove that if no further matches are possible at a node v, then l can be set to $c - lp(v)$ and the comparisons resumed at node n_v without missing any occurrences in T of patterns from P.
12. Prove that the search phase of the Aho–Corasick algorithm runs in $O(m)$ time if no pattern in $\mathcal{P}$ is a proper substring of another, and otherwise in $O(m + k)$ time, where k is the total number of occurrences.
13. The Aho–Corasick algorithm can have the same problem that the Knuth-Morris-Pratt algorithm

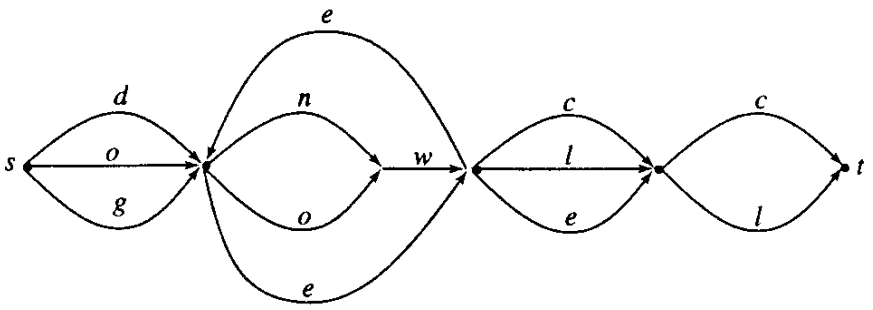

Figure 3.19: Directed graph for the regular expression $(d+o+g)((n+o)w)^*(c+l+\epsilon)(c+l)$.

aykkpqppvdtpq is a string specified by R. To specify S, the subexpression (p+q) of R was repeated four times, and the empty string ϵ was the choice specified by the subexpression $(l+z+\epsilon)$.

It is very useful to represent a regular expression R by a directed graph $G(R)$ (usually called a nondeterministic, finite state automaton). An example is shown in Figure 3.19. The graph has a start node s and a termination node t, and each edge is labeled with a single symbol from $\Sigma \cup \epsilon$. Each s to t path in $G(R)$ specifies a string by concatenating the characters of C that label the edges of the path. The set of strings specified by all such paths is exactly the set of strings specified by the regular expression R. The rules for constructing $G(R)$ from R are simple and are left as an exercise. It is easy to show that if a regular expression R has n symbols, then $G(R)$ can be constructed using at most 2n edges. The details are left as an exercise and can be found in [10] and [8].

Definition A substring T' of string T matches the regular expression R if there is an s to t path in $G(R)$ that specifies T'.

Searching for matches

To search for a substring in T that matches the regular expression R, we first consider the simpler problem of determining whether some (unspecified) *prefix* of T matches R. Let $N(0)$ be the set of nodes consisting of node s plus all nodes of $G(R)$ that are reachable from node s by traversing edges labeled ϵ. In general, a node v is in set $N(i)$, for $i > 0$, if v can be reached from some node in N(i − 1) by traversing an edge labeled $T(i)$ followed by zero or more edges labeled ϵ. This gives a constructive rule for finding set $N(i)$ from set N(i − 1) and character $T(i)$. It easily follows by induction on i that a node v is in $N(i)$ if and only if there is path in $G(R)$ from s that ends at v and generates the string T[1..i]. Therefore, prefix T[1..i] matches R if and only if $N(i)$ contains node t.

Given the above discussion, to find all prefixes of T that match R, compute the sets $N(i)$ for i from 0 to m, the length of T. If $G(R)$ contains e edges, then the time for this algorithm is $O(me)$, where m is the length of the text string T. The reason is that each iteration i [finding $N(i)$ from N(i − I) and character $T(i)$] can be implemented to run in $O(e)$ time (see Exercise 29).

To search for a *nonprefix* substring of T that matches R, simply search for a prefix of T that matches the regular expression C*R. C* represents any number of repetitions (including zero) of any character in Σ. With this detail, we now have the following:

Theorem 3.6.1. *If T* is of length m, and the regular expression R contains n symbols, then it is possible to *determine whether* T *contains* a substring matching R in $O(nm)$ time.

(i, j – i + 1). Then declare that P occurs in T with upper left corner in any cell whose counter becomes n (the number of rows of P). Does this work?

Hint: No.

Why not? Can you fix it and make it run in O(n + m) time?

27. Suppose we have q > 1 small (distinct) rectangular pictures and we want to find all occurrences of any of the q small pictures in a larger rectangular picture. Let n be the total number of points in all the small pictures and m be the number of points in the large picture. Discuss how to solve this problem efficiently. As a simplification, suppose all the small pictures have the same width. Then show that O(n + m) time suffices.

28. Show how to construct the required directed graph $G(R)$ from a regular expression R. The construction should have the property that if R contains n symbols, then $G(R)$ contains at most $O(n)$ edges.

29. Since the directed graph $G(R)$ contains $O(n)$ edges when R contains n symbols, $|N(i)| = O(n)$ for any i. This suggests that the set $N(i)$ can be naively found from N(i – 1) and $T(i)$ in $O(ne)$ time. However, the time stated in the text for this task is $O(e)$. Explain how this reduction of time is achieved. Explain that the improvement is trivial if $G(R)$ contains no ϵ edges.

30. Explain the importance, or the utility, of ϵ edges in the graph $G(R)$. If Rdoes not contain the closure symbol "*", can ϵ edges always be avoided? Biological strings are always finite, hence "*" can always be avoided. Explain how this simplifies the searching algorithm.

31. Wild cards can clearly be encoded into a regular expression, as defined in the text. However, it may be more efficient to modify the definition of a regular expression to explicitly include the wild card symbol. Develop that idea and explain how wild cards can be efficiently handled by an extension of the regular expression pattern matching algorithm.

32. PROSITE patterns often specify the number of times that a substring can repeat as a finite range of numbers. For example, CD(2–4) indicates that CD can repeat either two, three, or four times. The formal definition of a regular expression does not include such concise range specifications, but finite range specifications can be expressed in a regular expression. Explain how. How much do those specifications increase the length of the expression over the length of the more concise PROSITE expression? Show how such range specifications are reflected in the directed graph for the regular expression (ϵ edges are permitted). Show that one can still search for a substring of T that matches the regular expression in $O(me)$ time, where m is the length of T and e is the number of edges in the graph.

33. Theorem 3.6.1 states the time bound for determining if T contains a substring that matches a regular expression R. Extend the discussion and the theorem to cover the task of explicitly finding and outputting all such matches. State the time bound as the sum of a term that is independent of the number of matches plus a term that depends on that number.

has when it only uses sp values rather than sp' values. This is shown, for example, in Figure 3.16 where the edge below the character *a* in *potato* is directed to the character *a* in *tattoo.* A better failure function would avoid this situation. Give the details for computing such an improved failure function.

14. Give an example showing that k, the number of occurrences in T of patterns in set $\mathcal{P}$, can grow faster than $O(n+m)$. Be sure you account for the input size n. Try to make the growth as large as possible.

15. Prove Lemmas 3.4.2 and 3.4.3 that relate to the case of patterns that are not substring free.

16. The time analysis in the proof of Theorem 3.4.1 separately considers the path in $\mathcal{K}$ for each pattern P in $\mathcal{P}$. This results in an overcount of the time actually used by the algorithm. Perform the analysis more carefully to relate the running time of the algorithm to the number of nodes in $\mathcal{K}$.

17. Discuss the problem (and soiution if you see one) of using the Aho–Corasick algorithm when a, wild cards are permitted in the text but not in the pattern and b. when wild cards are permitted in both the text and pattern.

18. Since the nonlinear time behavior of the wild card algorithm is due to duplicate copies of strings in P, and such duplicates can be found and removed in linear time, it is tempting to "fix up" the method by first removing duplicates from $\mathcal{P}$. That approach is similar to what is done in the two-dimensional string matching problem when identical rows were first found and given a single label. Consider this approach and try to use it to obtain a linear-time method for the wild card problem. Does it work, and if not what are the problems?

19. Show how to modify the wild card method by replacing array C (which is of length m > n) by a list of length n, while keeping the same running time.

20. In the wild card problem we first assumed that no pattern in $\mathcal{P}$ is a substring of another one, and then we extended the algorithm to the case when that assumption does not hold. Could we instead simply reduce the case when substrings of patterns are allowed to the case when they are not? For example, perhaps we just add a new symbol to the end of each string in $\mathcal{P}$ that appears nowhere else in the patterns. Does it work? Consider both correctness and complexity issues.

21. Suppose that the wild card can match any length substring, rather than just a single character. What can you say about exact matching with these kinds of wild cards in the pattern, in the text, or in both?

22. Another approach to handling wild cards in the pattern is to modify the Knuth-Morris-Pratt or Boyer–Moore algorithms, that is, to develop shift rules and preprocessing methods that can handle wild cards in the pattern. Does this approach seem promising? Try it, and discuss the problems (and solutions if you see them).

23. Give a complete proof of the correctness and $O(n+m)$ time bound for the two-dimensional matching method described in the text (Section 3.5.3).

24. Suppose in the two-dimensional matching problem that Knuth-Morris-Pratt is used once for each pattern in P, rather than Aho–Corasick being used. What time bound would result?

25. Show how to extend the two-dimensional matching method to the case when the bottom of the rectangular pattern is not parallel to the bottom of the large picture, but the orientation of the two bottoms is known. What happens if the pattern is not rectangular?

26. Perhaps we can omit phase two of the two-dimensional matching method as follows: Keep a counter at each cell of the large picture. When we find that row i of the small picture occurs in row j of the large picture starting at position (i', j), increment the counter for cell

0	1
0	0
1	0
0	1
1	0
1	1
0	1
1	0

Figure 4.1 : Column $j-1$ before and after operation *Bit-Shift*$(j-1)$.

4.2.1. How to construct array M

Array M is constructed column by column as follows: Column one of M is initialized to all zero entries if $T(1) \neq P(1)$. Otherwise, when $T(1) = P(1)$ its first entry is 1 and the remaining entries are 0. After that, the entries for column $j > 1$ are obtained from column $j-1$ and the U vector for character $T(j)$. In particular, the vector for column j is obtained by the bitwise AND of vector ***Bit-Shift***$(j-1)$ with the U vector for character $T(j)$. More formally, if we let $M(j)$ denote the jth column of M, then $M(j) = Bit\text{-}Shift(j-1)$ AND $U(T(j))$. For example, if $P =$ ***abaac*** and $T =$ ***xabxabaaxa*** then the eighth column of M is

1
0
1
0
0

because prefixes of P of lengths one and three end at position seven of T. The eighth character of T is character ***a***, which has a U vector of

1
0
1
1
0

When the eighth column of M is shifted down and an **AND** is performed with $U(a)$, the result is

1
0
0
1
0

which is the correct ninth column of M.

To see in general why the ***Shift-And*** method produces the correct array entries, observe that for any $i > 1$ the array entry for cell (i, j) should be 1 if and only if the first $i-1$ characters of P match the $i-1$ characters of T ending at character $j-1$ and character $P(i)$ matches character $T(j)$. The first condition is true when the array entry for cell $(i-1, j-1)$ is 1, and the second condition is true when the ith bit of the U vector for character $T(j)$ is 1. By first shifting column $j-1$, the algorithm ANDs together entry $(i-1, j-1)$ of column $j-1$ with entry i of the vector $U(T(j))$. Hence the algorithm computes the correct entries for array M.